The Security and Performance Impact of Object File Shuffling

A Thesis

Presented in Partial Fulfillment of the Requirements for the

Degree of Master of Science

with a

Major in Computer Science

in the

College of Graduate Studies

University of Idaho

by

Jonathan Buch

Major Professor: Jim Alves-Foss, Ph.D.

Committee Members: Jia Song, Ph.D.; Daniel Conte de Leon, Ph.D.

Department Administrator: Terry Soule, Ph.D.

May 2020

Authorization to Submit Thesis

This thesis of Jonathan Buch, submitted for the degree of Master of Science with a major in Computer Science and titled **"The Security and Performance Impact of Object File Shuffling,"**has been reviewed in final form. Permission, as indicated by the signatures and dates given below, is now granted to submit final copies to the College of Graduate Studies for approval.

Major Professor: _____ Date:_____

Jim Alves-Foss, Ph.D.

Committee Members: _____ Date:_____

Jia Song, Ph.D.

_____ Date:_____

Daniel Conte de Leon, Ph.D.

Department

Administrator: _____ Date:_____

Terry Soule, Ph.D.

Abstract

Software running on desktop computers, laptops, and servers can be updated on a regular basis, installing security and bug fixes. However, industrial control system devices and embedded devices are often deployed and then left in operation for long periods of time with no software updates. The software running in these devices is often installed as an integral part of the system, and is typically called firmware. As these devices age, many have security vulnerabilities found that are located in their firmware or related libraries and as such need to be patched to mitigate the vulnerability, or they are otherwise vulnerable to exploitation. One issue with updating this firmware is that the original image may contain an old version of a library that they rely on for their processes and haphazardly updating may break that functionality. Also, updates may cause changes in critical real-time behavior of the systems. If the firmware is not updated, attacks that exploit discovered vulnerabilities can be successful against all of the deployed devices. This research explores diversification of deployed firmware through the use of shuffling portions of the firmware's code while retaining original functionality. This thesis examines the security impact of shuffling and then reports on a set of experiments that look at performance impact of the shuffling. Results indicate that shuffling can improve security against many modern low-level attacks, and that rearranging the code can change run-time performance of the program by a couple percentage points. With increased security and little performance impact, we recommend further study into the use of shuffling as an added security mechanism.

Acknowledgments

I would first like to thank my advisor and major professor, Dr. Jim Alves-Foss, for his support, encouragement, and patient guidance throughout my graduate studies.

I would also like to thank my other committee members, Dr. Daniel Conte de Leon and Dr. Jia Song, for their valuable comments and critiques on my thesis.

I would like to thank all my instructors for their hard work and dedication in providing me with a comprehensive and valuable education.

I wish to acknowledge the National Science Foundation CyberCorps®, for supporting me during the course of my graduate studies through grant number DGF-1565572.

Last, but certainly not least, I would like to thank my family and friends for their understanding, support, encouragement, and love which have helped me make this thesis a reality.

To my dear parents and sister...

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Firmware, loosely defined, is the software that is installed inside of devices to control how they operate. This software is required by the system/device and generally not under the control of an end user. Firmware is also updated much less frequently than most software, if at all. Firmware itself comes in many different forms whether it is used in an embedded system like a Smart TV or being the basis of how the operating system runs on your home personal computer or smart phone. With a wide variety of completely different flavors of firmware, its difficult to both determine what security protections firmware should be using and to what extent protections should be implemented while still retaining a reasonable overhead cost. In addition, firmware developers are not usually trained in security or their software has memory and time constraints that make the inclusion of security features very difficult. This has all led to a limited set of security features in firmware.

1.1 Problem Space

Some of the more recent attacks against firmware show just how vulnerable the firmware that devices are running nowadays. These include attacks such as the Mirai Botnet and Stuxnet, both of which target low level devices [AAB+17, BBLD13]. The Mirai botnet was an attack in 2016 that utilized a worm type malware called Mirai. Mirai functioned by scanning the network that an initially infected device and looked for devices that were running telnet or SSH. From there, it would attempt to use the default credentials of internal library of hard-coded Internet of Things (IoT) credentials. Since many of these embedded devices had unchanged credentials, the worm was able to spread like wildfire. Once infected, the IoT devices received commands to issue more scans and infect more devices while also being able to target specific to enact a distributed denial of service (DDoS) attack using all of the traffic generated by the embedded

devices. A DDoS attack is meant to target the availability of the victim by overloading the bandwidth with traffic from many different devices. Due to embedded devices being simple in nature and many using default credentialls, they were the perfect targets for enabling the botnet to be as successful as it was. The Mirai Botnet showed that the security of embedded devices and firmware in general needs to be developed with security in mind in order to stop any future attacks from being successful.

This has led to security mechanisms such as Address Space Layout Randomization (ASLR) being implemented within firmware, as an attempt to thwart these exploits. ASLR allows programs to be placed at different locations in memory, hopefully thwarting attacks that rely on knowledge of actual memory addresses. Unfortunately, ASLR mechanisms are limited and can be prone to attacks [LHG+11, SPP+04, SD08, Ser12] as discussed in Chapter 2. One of the concerns with ASLR is that all of the code is shifted in memory as a single block. One improvement to ASLR is to use a more fine-grain reorganization of the code, via shuffling. Shuffled code limits the ability of the attacker to know relative locations between two pieces of code. There are multiple different forms of shuffling, based on either the granularity of the shuffling, or when the shuffling takes place. Since programs contain instructions that reference other instructions or data code either needs to be compiled as a position independent executable (PIE) or contain relocation tables that are populated when the program is stored in memory. Further details of these issues and this process are provided in Chapter 4.

The granularity of shuffling can occur at major memory sections level (the stack, heap, and code) which is what ASLR does, or the code can be further shuffled at the granularity of object files, functions, or basic blocks (where a basic block is a sequence of instructions that are always executed together – having a single entry point and a single exit point). This shuffling can occur when the file is first created, at compile and link time, at installation time as the program is loaded into memory, or dynamically as it is running. To obtain diversity for a program that is shuffled at compile and link time, there must be a process to distribute different versions of the same executable to each customer. If shuffling occurs at installation time, the installation mechanism must have a way of randomizing the program as it is installed. This can occur by distributing the program as a package of separate object files, functions, or blocks, with appropriate relocation information, and a process of combining them and updating any necessary reference addresses – similar to what is done in the final stages of compilation. If the shuffling occurs at load time, then the relocation information must be maintained in the program

and used by the loader. This process of loading the actual addresses relative to a relocation table was historically used before the widespread adoption of separate virtual address spaces for processes, and could be included in the system loader. Each of the previous mechanisms is increasingly less static. The final shuffling mechanism involves dynamically shuffling the process while it is running. This involves copying the process into a set of new addresses and updating all of the links, and then transferring control to the new copy [WKGWK+16]. This process also requires all relevant relocation information for hard-coded or relative address references. This dynamic shuffling requires substantially more resources in terms of memory and processing power than any of the other mechanisms.

Currently, for the majority of devices, the exact same firmware version is usually installed across each instance of the same device. For example, in a power station with hundreds of the same type of relay, each is running the exact same firmware image. This is an issue because an exploit that is developed to work on the specific firmware image can be leveraged against every device running that same firmware image. As a result, every device is vulnerable to the same exploit and the adversary can choose whichever device to use the exploit on, eventually being able to exploit them all.

1.2 Motivation

The primary goal of this research was to determine the security impact of finer granularity shuffling and to determine the security and performance impacts of such shuffling. Of particular note is the concept that the ordering of object files or functions may impact performance behavior due to caching or other hardware optimizations. Therefore, this project focusing on non-dynamic shuffling and specifically looks at shuffling prior to loading. A companion research project is exploring the concepts of shuffling during load time. In addition, to avoid the necessity of modifying a compiler or the source files in the test suites, this project focuses on the impact of object file shuffling. Fig. 1.1 is a visual representation of how object file shuffling works in the context of this project.

Shuffling helps mitigate attacks by generating unique firmware versions using the same code base, ensuring that functionality is retained in the new versions. This method allows each device to be running different, yet functionally identical, firmware images. Using this method, an attack developed to be effective on one device will be ineffective on another of the same type. The exploit is unable to propagate between all the same device by a simple modification

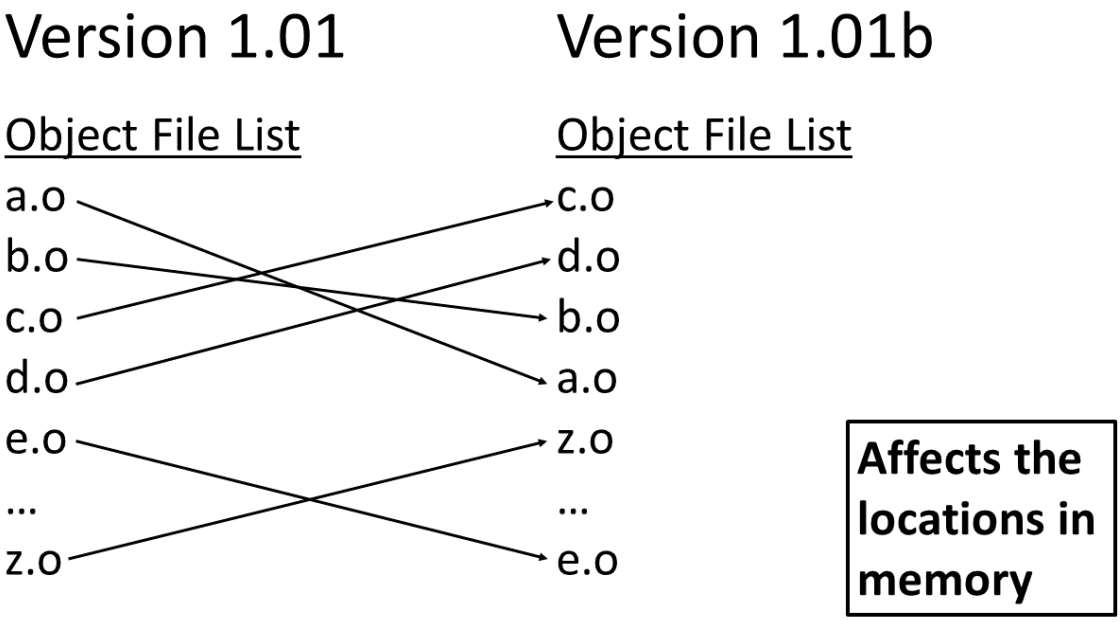

Figure 1.1: Object File Shuffling Visualized

of the order that object files are linked. Fig. 1.2 shows a possible scenario in which a version of firmware, version 1.01, would be split into several different, yet functionally similar, versions: 1.01a, 1.01b, 1.01c, and 1.01d.

The aim of this research is to determine the effect that shuffling the order of the functions in an executable has on the performance of the system and its security benefits. To that end, this project has the following specific objectives and tasks.

1. Objective 1. Evaluate the security implications of shuffling
   (a) Task 1. Literature search and summary of different types of memory rearrangement techniques
   (b) Task 2. Literature search and summary of attacks against memory rearrangement
   (c) Task 3. Summary of how function or object file shuffling can mitigate the Task 2 attacks and is better than Task 1 approaches
2. Objective 2. Develop a technique for shuffling executables
   (a) Task 1. Explore different shuffling mechanisms including basic block shuffling, function shuffling, and object file shuffling and determine strengths and weaknesses of each

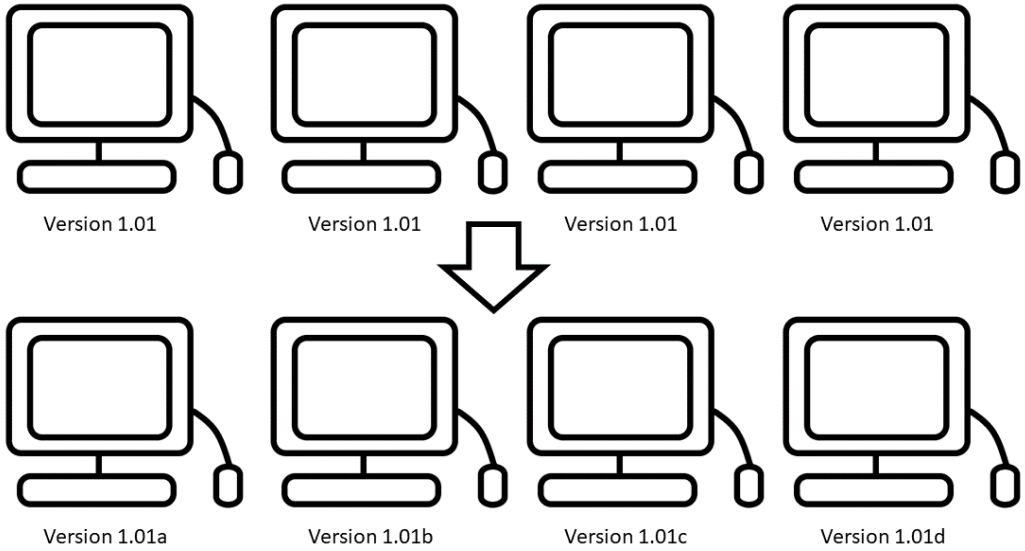

Figure 1.2: Unique Version Creation via Shuffling

(b) Task 2. Implement a shuffling mechanism

3. Objective 3. Evaluate the performance impact of shuffling

   (a) Task 1. Run experiments on multiple test suites

   (b) Task 2. Analyze the results

1.3 Thesis Impact

This research's impact helps determine the security viability of object file shuffling and evaluate its performance relative to other available and similar security alternatives such as ASLR and stack canaries. Additionally, evidence shows that function shuffling can be used as an effective and efficient solution to creating diverse yet functionally similar executables from the same source code. This allows unique firmware versions to be created without fundamentally changing or modifying the original code base. The amount of object files the executable is dependent on will determine how many different versions of the same executable can be created.

As a result of this versioning method, a set of test suites for new security methods was developed based on SPEC CPU® 2017 [Cor20]. The newly generated test suites can provide more information on what kind of performance impact potential security methods have on a defined system. This research contributes to the scientific community by:

1. Providing an exploration and analysis of shuffling as a mitigation technique. This will help those in research and industry that are looking for methods to improve firmware security.

2. Providing a method in which test suites can be artificially expanded by the use of object file shuffling. This will help those that are developing new security methods and techniques by offering additional testing capabilities.

This research can be used by those that are reliant on the firmware of a device of which they have many units of and have critical functionality. One such type of device would be relays within a power station.

1.4 Thesis Overview

The remainder of this thesis is organized as follows.

- Chapter 2 provides background on the current firmware security methods which are currently in use or have been proposed by other researchers and developers.

- Chapter 3 provides detailed discussion of the technologies used in our approach, and the prior work we are building on.

- Chapter 4 evaluates the potential security impact the addition of shuffling provides. Additional detail is provided on the enabling features of some well known attacks as well as information on other security techniques' effectiveness.

- Chapter 5 presents the experimental designs and methods used for developing the ability of object file shuffling.

- Chapter 6 presents the the analysis of the performance impact of object file shuffling using the SPEC 2017 Benchmark.

- Chapter 7 presents conclusions and insight gained from the performance and security impacts of object file shuffling. Future direction is laid out to extend and expand the work.

Chapter 2

Survey of Current Firmware Security Methods

This chapter summarizes several different techniques that are currently used to secure firmware. Since firmware is varied, especially with respect to security approaches, this chapter discusses some of the more common techniques. For each of the methods, this chapter evaluates the level of protection provided, weaknesses and ways the mechanisms can be bypassed.

2.1 Address Space Layout Randomization

Address Space Layout Randomization (ASLR) is a method in which the base addresses of an executable are randomly assigned in memory during the loading of the executable as well as during the linking process [SPP+04]. This includes the addresses of the stack, heap, and code. This way, each time the executable is run, it has different memory address locations for each part of an executable. This includes any included libraries and other parts of the executable.

2.1.1 What is ASLR's effect on security?

ASLR is primarily used to protect the addresses of the functions on the stack within an executable. Since adversaries use the locations of certain functions to craft exploits, this method helps in removing the ability for the attacker to find those addresses. By randomizing each of the locations, an adversary cannot be certain of the exact locations of the functions consistently and are therefore unable to carry out those exploits. The most common exploit that ASLR is used to mitigate is a buffer overflow. A buffer overflow occurs when a buffer size is miscalculated either by accident by the developer or some other error occurs while creating the code and a larger chuck of memory is copied into a smaller buffer, thus overrunning other data within the memory. When an adversary attempts to exploit a buffer overflow, they aim to overwrite the return address of the function that the buffer overflow occurs in in order to return

to code that the adversary selects or controls. Normally the target code is specially crafted code called shellcode that the adversary has made specifically to gain control [Ale96], or, more recently, a chain of returns to select instructions and functions called return oriented programming (ROP) [RBSS12]. While ASLR is a good baseline security solution, it has been shown that there are certain attacks that can bypass ASLR. The presence of these attacks means that ASLR by itself is not an effective strategy. However, since ASLR is simple and effective and can be implemented by the operating system, it can be easily combined with other security methods such as a stack canary in order to provide better overall security.

2.1.2 How is ASLR implemented?

ASLR is concerned with the 3 areas of the assigned user address space: the executable code, the mapped area, and the stack area [SPP+04]. The initialized and uninitialized data along with the executable code are housed within the executable area. The heap and other dynamically allocated data are located within the mapped area. Lastly, the stack contains the user stack. As stated previously, each of the areas is randomly assigned a start address within the executable's user space. The value that each of the areas are offset by is dependent on the architecture that the executable is running on. For Intel 32-bit x86 processors, ASLR randomizes with 16 bits of randomness for the executable and mapped areas while the stack gets 24 bits of randomness [SPP+04]. There are two reasons why there is a limited amount of randomness, particularly for the mapped area offset, `delta_mmap`. First, randomizing bits 28 through 31 would inhibit `mmap()` system call's handling of large memory mappings. The lowest 12 bits of memory address specify a location on a 4K memory page. Since compilers will often optimize placement of code based on page boundaries, changing the bottom 12 bits causes memory mapped pages to not align to page boundaries. Therefore, ASLR can only modify the middle 16 bits. For 64-bit machines, there is usually no less than 40 bits of randomness which leads to brute force attacks being deemed as infeasible [SPP+04].

2.1.3 What are the weaknesses of ASLR?

With randomized addressing, there are some issues in the form of tracing and general debugging of the executable. Since it is randomized, it becomes harder to track exactly where an issue occurs as it changes each time the executable is loaded into memory, whereas if ASLR wasn't enabled it'd be much easier to see that an error only occurs the same place consistently. Additionally, there are several attacks that have been shown to defeat ASLR [MGR16,MGR14,EPAG16].

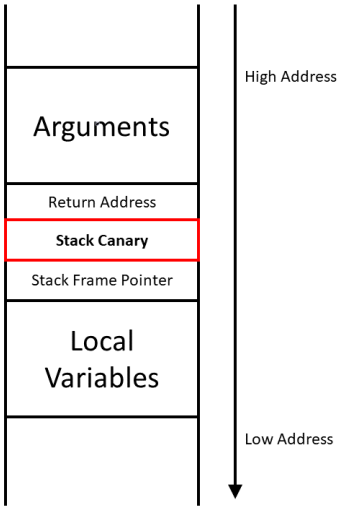

Figure 2.1: Stack Canary Location

Marco-Gisbert et al. [MGR14] discuss how the low entropy of ASLR on 32-bit systems is enough to make brute forcing an address of a library much easier than expected. Entropy in this case refers to the predictability of where the objects will be placed in memory. Since objects have certain ranges where they can be placed, the entropy varies between each object. Higher entropy means a longer time for the adversary to obtain a location from which they can determine the location of other sections of the executable in memory. If ASLR fails and there are no other countermeasures in place, the attacker will be able to do a variety of things, most notably gaining a shell with elevated privileges on the system. From there, the adversary can do whatever they would like to the system since they have the privileges to do so.

2.2 Stack Canaries

A stack canary is a value placed on the stack between the return address of the current function and the local variables. This value can be anything as long as it is known by the system in order to be checked. This value is checked while the program is running and if it has been observed to be changed or modified, the program will terminate. Stack canaries first came about in the gnu compiler, gcc, in 1998 with the introduction of StackGuard [WC$^+$03]. StackGuard describes the location of a canary as the location directly after the control information which is typically a return instruction within every frame. It further describes the different types of canaries which all have similar protections but differ in what they detect. Fig. 2.1 shows the location of the stack canary in memory and how it serves as a sort of barrier to the return address.

2.2.1 What is the effect of stack canaries on security?

A stack canary has several security benefits, most notably by defeating attempted exploits seen in programs such as buffer overflows. Since the canary value can be checked if it has been changed or not, it is a simple but effective method at determining if a buffer has been written outside of its allocated space and onto the stack. Another positive aspect of a stack canary is that it can be combined with other security techniques to enhance the security of the program even further. Most typically, stack canaries are used in conjunction with ASLR in order to better defend against buffer overflows and similar attacks. Since the location of the canary is dependent on where the control information in a frame is, ASLR doesn't have a negative effect when used together as the canary location is calculated after the program is in memory.

2.2.2 How are stack canaries implemented?

Stack canary implementation is architecture dependent. StackGuard describes three different types of canaries: terminator canaries, random canaries, and random XOR canaries. Terminator canaries have a value comprised of the termination characters including NULL and -1. This prevents standard string operations, which stop at NULL, from overwriting the canary and values past it. Random canaries are the typical canary which contains a 32-bit secret value. Random XOR canaries utilize a combination of a random canary coupled with using exclusive or on the control information it is protecting and setting the result as the canary value [WC$^+$03].

2.2.3 What are the weaknesses of stack canaries?

While it is a simple and effective method of adding additional security to a program, stack canaries do have some downsides. The first and foremost is that if the adversary is able to read the value of the canary and input the value in the buffer that is being overflowed in the correct location where the result of the buffer overflow places the same value in the correct location, the stack canary check will erroneously succeed and the buffer overflow won't be detected. Another strategy that is effective as shown by [Ric02] is overwriting the Global Offset Table (GOT) entry for different functions; in their example using the __exit() function address and gaining control over the execution flow of the program. Much like when ASLR fails, when a stack canary fails, it allows an adversary to modify execution flow of a program to their own addresses and shellcode.

2.3 Non-Executable Stack

A non-executable stack is the general term used for marking the memory where the stack is located as non-executable. A non-executable stack is used as a defense against attacks that are able to place malicious code on to the stack to be executed, the most common of which is done via buffer overflow. Removing the ability for code to be executable on the stack ensures that even if code were to get injected, it wouldn't be successful in executing and accomplishing the adversary's goal. Data Execution Prevention is Microsoft's implementation of the non-executable stack while including some other protections. For Linux, the non-executable stack was included in the PaX package in 2001 along with the inclusion of ASLR [MGR16].

2.3.1 What is effect of the non-executable stack on security?

A non-executable stack helps protect against attacks that ASLR is unable to defend against as ASLR is only concerned with the adversary having difficulty finding the locations of libraries and other information and not how the attacker exploits the sytem to jump to those locations. A non-executable stack working in conjunction with ASLR both makes it difficult for the adversary to find important addresses while also making it impossible to execute their injected code on the stack.

2.3.2 How is a non-executable stack implemented?

The non-executable stack is implemented via an option within /etc/system for Solaris 2.6 and later. Additionally, the implementation of the non-executable option adds no additional overhead to running executables as well as not requiring recompilation [CBD+99]. This way, the non-executable stack protection can be enabled with other protection mechanisms without interfering with other mechanisms put in place. Even combining the non-executable stack with the similar StackGuard protections works with no issues [CBD+99]. A non-executable stack does require hardware and OS support.

2.3.3 What are the weaknesses of a non-executable stack?

As it only affects the stack, the heap is left wide open to similar exploits. Cowan et al. [CBD+99] detail how one of the potential vulnerabilities is having the attacker inject shellcode within the heap buffer and then use the stack buffer to overwrite the return address to the location in the heap where the shellcode is located. So, as a result of this effect, a non-executable stack unfortunately cannot prevent every variation of the attack it is meant to protect. Microsoft's

DEP solves this issue by marking data pages non-executable, effectively able to cover both the stack and the heap. Unfortunately, there are some attacks that can get the operating system to change permissions on the stack, making it executable. For Just in Time compiled code, this method is ineffective since the newly compiled code is placed on executable data pages.

2.4 Data Execution Prevention

DEP is a system used by Microsoft to add in memory protection by giving the ability of defining pages of memory as non-executable. This includes the non-executable protections mentioned earlier. This essentially makes it so when a buffer overflow vulnerability is exploited, the overwritten bytes are unable to be executed since they are in a section of memory marked as non-executable by DEP. DEP uses a exception handling status code called STATUS_ACCESS_VIOLATION where, if it isn't handled, terminates the process [Sat18]. By default, applications cannot execute code on the stack nor the heap while DEP is in place and space allocated by malloc() is automatically set as non-executable. There are a few cases where you can specify memory locations where you can give execution privileges, but those require specific designations in order to grant those permissions. Also, as an additional rule, a memory page cannot be made both writable and executable [MSL12]. This is accomplished through a value known as the NX bit. The value of this bit determines if a page is marked as non-executable. With this designation, if an adversary is able to write inside a memory page, that page is already designated as non-executable so any shellcode that the adversary injects is unable to be executed and the exploit would fail.

2.4.1 What is Data Execution Prevention's effect on security?

DEP effectively stops code from being executed within non-executable, writable memory pages but it isn't an end all solution as it still has some issues with other types of vulnerabilities and exploits. For example, some resources have shown that with DEP as the only active security method in place, attacks such as return to libc [Sha07] and return oriented programming [RBSS12] gadget chaining are still viable means of exploiting the program. However, much like how ASLR and stack canaries can be combined to create an even more secure memory space, DEP can also be combined with these methods in order to prevent those attacks. With the combination of DEP and ASLR, you are able to make memory non-executable while still stopping attacks such as return to libc thereby increasing the overall security of the program.

2.4.2 How secure is Data Execution Prevention?

Much like with the non-executable stack for Linux, DEP is also vulnerable to several different types of attacks [SGGK07], [CFBX11]. These attacks include Return-into-Libc and Just in Time spraying (JIT spraying) [CFBX11] among others. Return to libc involves using the functions within the C standard library, libc, as it is loaded into nearly every program [Sha07]. Using the functions provided by libc, attackers can utilize well know calls and routines to craft an exploit on anything that includes the library. According to Microsoft, DEP isn't compatible with all functionality so its inclusion in program protection could be limited to none depending on what functionality fails while DEP is enabled.

2.4.3 What are the weaknesses of Data Execution Prevention?

DEP may cause some issues when an executable uses the stack or heap for execution of some of its functionality. Since the memory pages are marked as non-executable, those executables may have limited to no functionality while running with DEP enabled. This is mainly an issue with older programs that were developed before the introduction of DEP. An additional side effect of DEP interfering with execution is that a user may shut off the protection as a means of getting functionality restored rather than making an exception case for that particular executable.

2.5 Summary

The main takeaway from all of these mitigation techniques is that they are much more effective when paired with each other rather than on their own. If mitigation techniques are able to be enabled together without compromising the functionality of the program or adding too much additional overhead and are able to be combined without breaking the other, then it is a net benefit to the security of the program to have them collectively enabled. That way, if one of the methods isn't capable of stopping one type of attack but is effective at stopping a separate but just as serious attack, the combination of the methods would ensure that the program is overall more secure. Overall, there are a variety of different security techniques that can be utilized, including ASLR and DEP, and the effective implementation and combination of them can improve the security of programs as a whole. Unfortunately, exploits have been developed that bypass each of these techniques, even when use in combination. We discuss these exploits and how they are addressed by shuffling in Chapter 4.

Chapter 3

Background

This chapter provides an introduction to firmware and firmware security. The major parts of firmware creation and other relevant topics are explored. This includes Position Independent Code, Position Independent Executables, the Linker and its processes, and the Executable Format and Addressing.

3.1 The Linker and the Linking Process

The linker is the last part of the compilation process and is critical to ensuring all of the different parts of an executable come together efficiently and correctly. It is responsible for making sure all of the specified libraries and dependencies are all accounted for and also for assembling the final executable along with any supplied options that the user may give it. Some of these options like –fPIE and –fPIC determine how addresses are handled and what will go into the executable.

3.1.1 Types of Linking

There are two types of linking processes, static linking and dynamic linking. Static linking creates the executable with all of the specified code with no outside references or external libraries required. This allows the executable to run on its own without extra code from elsewhere in memory. Since all library functions that are referenced in the code have already been copied in, referencing is much faster but at a cost of having a larger file size [Cen20]. Dynamic linking, on the other hand, uses references for functions that are determined at load time. Any shared libraries involved can be used by multiple different processes from one location. Since the shared libraries are dynamically linked in and are located outside of the code itself, the executable size is much smaller than it would have been if it had been statically linked. Also, updates to the library will not require recompilation. However, since the dynamically linked functions are not

located within the executable in memory, the calls made to them are slower and have a larger overhead. Shared libraries also aren't loaded into memory until they are referenced, thereby increasing the overhead cost [Cen20]. Dynamic linking also has an added issue that changes or updates that occur to dynamically linked libraries could cause the executables that are dependent on them to not function properly or at all in some cases unless the executable itself is updated as well to incorporate the new changes. In well-matured libraries, this is mitigated by using library versioning. Statically linked libraries do not have this problem due to the library functions already existing in the executable itself.

3.2 Global Offset Table

The Global Offset Table is a table of addresses populated during the run time of an executable defining the locations of global variables including the function addresses of shared libraries. Since absolute addressing cannot be assumed to be completely removed from the code prior to compilation, the GOT houses these addresses for the respective source. Each library and module linked into an executable will have its own GOT entry referencing those absolute locations and use relative addressing to reference those locations [ZDWZ03]. For example, replacing the entry to _exit() with the program _start location forces the program to keep running with the same ASLR offsets. If an attacker can do this and leak addresses, they can eventually map the process in memory.

3.2.1 Vulnerabilities

Since GOTs have location and data flow information contained within them, they are a prime target for adversaries when considering where to divert the execution flow to their control. Some research has been done on effective attacks on GOT which includes GOT hijacking. GOT hijacking involves locating the GOT's address in memory and changing the function pointers [ZDWZ03].

3.3 Position Independent Code

Position independent code (PIC) is code that is generated by the compiler that is able to be executed from any address in memory where it is placed. This differs from normal compilation absolute addresses for where jump locations are hardcoded into the executable which would result in errors when trying to move objects around in memory. Typically, jumps are hardcoded

within the executuble. There are two ways that they can be handled. The first is to use relative addressing. This involves changing any hardcoded address to an offset relative to where the jump is supposed to lead to. A side effect of relative addressing is that the code must move together in order to ensure those relative offsets are correct. The second way is to have a lookup table where those addresses can be resolved to their real locations. For PIC, this lookup table is the GOT. Since the addresses are stored in a lookup table, the code doesn't have to move as one single unit and can instead be shuffled. For gcc, the option to compile code into PIC is –fpic and –fPIC. One other aspect of PIC is that is can be set as read only code since it doesn't need to be patched by the loader [SDA02]. Position independent executables (PIE) are, on the other hand, closely related to PIC but there are several differences. First, PIE programs are, as the name implies, executables compiled to be position independent. This means that the resulting PIE can be placed anywhere in memory and function normally just like PIC. In fact, for gcc, PIE are compiled using generated PIC. Going further, generated PIC for PIE using the options –fpie and –fPIE can only be used for linking into executables [Fre09]. PIC compiled by itself without the –fPIE option has the additional capability of being used for other purposes such as shared libraries or shared objects. PIE are also similar to shared libraries as they share the fact that the dynamic linker is responsible for relocating them to the location in virtual memory where the OS puts them [CT91].

3.3.1 Security Impact

PIC and PIE offer a security benefit similar to ASLR. That is, they make it much harder for an adversary to locate specific places in code since the location of jumps and addresses are different each time the associated executable is run. Since the GOT is resolved at runtime, the addresses are able to be placed wherever the dynamic loader decides to place them. This results in being an effective defense against attacks such as ROP gadget chaining. ROP gadget chaining involves locating small sections of code that do certain actions, known as ROP gadgets, within memory. The ROP gadget addresses are then chained together in such a way that the end behavior is unintended by the developer and can exploit the executable. Since this type of attack is entirely reliant on locating the addresses of specific small sections of code to make the attack succeed and shuffling essentially randomizing those addresses each and every time the executable is run, it makes the adversary's job of finding those locations much more difficult and time consuming.

3.3.2 Downsides

Due to PIC and PIE not using absolute addressing, execution time ends up being a bit slower as there isn't a static address for which repeated lookups can occur efficiently [FCS+15, Pay12]. Schwarz et al. [SDA02] also describe three other problems with the position independent jump tables that normally come along with PIC which are: offset table appearing in the instruction stream, complex indirect jump patterns that are difficult to follow during review, and lastly the table not having relocation entries. These issues all correlate to being both more difficult to debug as well as being more difficult to perform code auditing on, but at the same time also make the adversary's job more difficult so it ends up a catch 22.

3.4 Memory Layout and Organization

It is useful to understand how processes like the linker or ASLR work, memory is laid out and where pieces of an executable are typically located. For this explanation, the memory layout of C will be used as its one of the most common memory layouts used in computing.

3.4.1 Sections of Memory

There are several sections that are defined for the memory layout. They include: the text section, the read only data, the block started by symbol (BSS) section, the stack, and the heap. The text section contains the machine language instructions for the program. The data section contains the initialized variables, both the global and static ones. The BSS section contains any uninitialized static or global variables. The stack contains the context of function calls as well as any local variables for each function. Lastly, the heap is responsible for any dynamically created variables during runtime using functions like malloc or calloc. There can be multiple heaps present depending on how much dynamically allocated there is in the executable.

Fig. 3.1 shows that, starting from the low addresses, the sections are normally arranged in the following order: text, data, BSS, heap, and finally stack. From where the heap starts, when new variables are dynamically allocated, they will be placed at a higher memory address than the previous. On the other hand, when pushing data onto the stack, they are assigned lower addresses from the previous value that was pushed on. This way, the heap "grows" toward the higher memory addresses while the stack "grows" towards the lower ones.

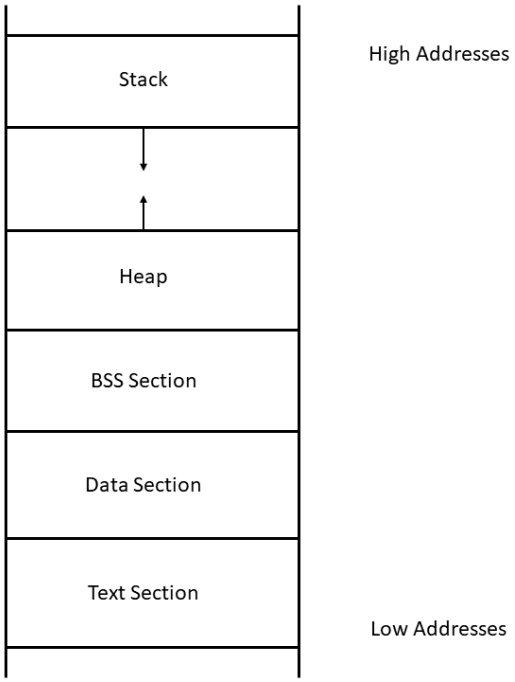

Figure 3.1: Layout of Memory

3.4.2 Overflows and their types

As discussed previously, there are several vulnerabilities related to how the memory is laid out. The most prevalent of which is a buffer overflow. There are several different types of buffer overflow: stack overflows, heap overflows, and data overflows. As their names imply, they are associated with an overflow in their respectively named section. With respect to stack overflows, they deal with the stack frame and how each of its individual sections are placed. The four major parts of a stack frame are the arguments, the return address, the previous frame pointer, and the local variables. The important part here is the location of the return address. Without a stack canary and other protections, the return address is vulnerable to an overflow of user controlled data and therefore control over the control flow could fall to the adversary.

3.5 SPEC CPU® 2017

SPEC CPU® 2017 [Cor20] is a tool that is used for testing performance of a system with various executables. It is designed to test hardware performance, but by using the same hardware, it can be used to test the performance of software changes. It comes with 4 test suites and 43 total benchmarks. The benchmarks provide a comparative measure of compute-intensive performance across the widest practical range of hardware using workloads developed from real

user applications [Cor20]. Due to this, the SPEC CPU® 2017 is the perfect testing ground for doing tests on performance of security methods and optimizations. The 4 test suites included are: Integer Rate (intrate), Integer Speed (intspeed), Floating Point Rate (fprate), and Floating Point Speed (fpspeed). The Rate test suites, intrate and fprate, are used to test the throughput of the system. The throughput is defined as the amount of work per unit of time. In other words, it measures the efficiency of each of the benchmarks. The Speed test suites, intspeed and fpspeed, instead measure the amount of time needed per benchmark. As its name implies, a higher score for these benchmarks indicates that less time was needed to run the benchmark. For each of the four test suites, there are a number of benchmarks provided, ranging from 10 to 14 benchmarks depending on which test suite is being used. Each of the benchmarks are executables that are practical applications. For example, modified versions of both the Perl Interpreter and the GNU C compiler are part of benchmarks as 500.perlbench_r and 502.gcc_r for intrate and 600.perlbench_s and 602.gcc_r for intspeed. Each of the benchmarks has a primary language which is either C, C++, or Fortran. Due to some initial difficulties compiling the Floating Point test suites fprate and fpspeed, intrate and intspeed were the test suites chosen for our experiments. The impact of our changes primarily involve the impact on checks and memory reference, so the use of integer benchmarks is sufficient.

Chapter 4

Security Analysis

4.1 Mitigation Effectiveness

This chapter details the security impact that shuffling has on executables. It discusses some of the potential exploits that could be defeated or made much more difficult to implement. Also discussed are other mitigations and their effectiveness on similar memory corruption attacks.

4.1.1 Preventative Measures

As a method of changing addresses of locations within an executable, shuffling gives protection against attacks that are reliant on finding and utilizing addresses in memory, much like ASLR. This type of mitigation is particularly effective against attacks known as return oriented programming (ROP). ROP attacks are reliant on little snippets of code in memory that, when put in a certain order, create malicious behavior. This type of attack is called ROP gadget chaining as the code snippets are the "gadgets" and are chained together to produce an undesired effect. ROP gadget chaining relies on locating each of the addresses of the gadgets that are in memory and placing them correctly in order. Due to its reliance on memory locations, the addition of object file shuffling should be an effective countermeasure. The reason why it is effective is that the addresses of the gadgets in one version of the shuffled binary will be different than those of a separately shuffled binary or even the original version, assuming the optimization level allows this behavior. Unlike ASLR, the relative positions of gadgets also changes, unless thy are in the same object file. Therefore an attack that is developed to work on a device that an attacker has possession of will be ineffective on the same device located somewhere else, thereby greatly weakening the scope of an attack to just one device. ASLR also changes ROP addresses. However, researchers have created new exploits to learn the appropriate addresses or use known offsets to calculate the addresses.

Table 4.1: Attacks Against Common Protection Mechanisms, credit to Varsha Venugopal

Protection Mechanisms	**Attacks**			
	Bypass ASLR, NX, Stack Canary [Shr18a]	Overwrite GOT table [Shr18b]	Return2libc to bypass NX bit [Shr18d]	Bypass NX bit using Mprotect function/Make stack Executable [Shr18c]
ASLR (Address Space Layout Randomization is a memory-protection process for OS that guards against buffer-overflow attacks)	✓	✓		
Stack Canary (Secret value placed on stack used to detect a stack buffer overflow before execution of malicious code)	✓			
NX bit (Prevents buffer overflow attacks by blocking code execution from memory that is marked as non-executable)	✓	✓	✓	✓
RELRO- Relocations Read Only (Security measure which makes some binary sections Read Only. Partial-Eliminates the risk of buffer overflows on a global variable overwriting GOT entries. Full- Makes the entire GOT read only)	Full	Partial	Full	Full
Fortify (Provides runtime checks of buffer lengths and memory regions)	✓	✓	✓	✓
PIE- Position Independent Executable (It loads binary into random locations within virtual memory each time the application is executed)	✓		✓	✓

Table 4.2: Enabling Features, credit to Varsha Venugopal

Enabling Features	**Attacks**			
	Bypass ASLR, NX, Stack Canary [Shr18a]	Overwrite GOT table [Shr18b]	Return2libc to bypass NX bit [Shr18d]	Bypass NX bit using Mprotect function/Make stack Executable [Shr18c]
Offset2libc (Offset of libc base found by dumping contents of stack using string format specifiers/using address of libc functions such as write/puts/read)	✓	✓	✓	✓
One_gadget offset (offset of one gadget from libc base address to call execve)	✓	✓	✓	
Offset to setuid (setuid function is found in libc which is set to 0, to get root privileges while executing remote shell)	✓		✓	✓
Pop Rdi value (It is a Rop gadget used to execute 'sh'. Address of 'sh' should be in rdi register)	✓		✓	✓
Address of _start function (This address is used to restart the program without letting the program to die)		✓		
Address of exit_got (This is usd to overwrite the GOT entry of exit() to start of the program whenever exit is called to prevent program from being killed)		✓		
Offset to /bin/sh (Offset of /bin/sh in libc to execute shell)			✓	
Offset to mprotect function (Mprotect function offset in libc is used to change the permissions of stack to make stack executable)				✓
Offset to memcpy (The offset to memcpy fucntion in libc is used to copy the shellcode from stack to different area in memory)				✓
Pop rdx, pop rsi ROP Chain (It is a Rop chain used to execute 'sh')				✓

4.1.2 Attack Explanation

In a companion project, conducted by Venugopal [Ven20], several attacks were evaluated with respect to modern security mechanisms. The preliminary results of this project are listed in Tables 4.1 and 4.2. The security mechanisms explored were ASLR, stack canary, and non-execution memory pages (NX). As discussed in Chapter 2, a combination of security mechanisms should improve the security of a system. This project explored that belief, utilizing attacks published by other researchers, as summarized in Table 4.1. Each of these attacks assumes some base knowledge, some base set of security controls, and possibly other configurations. Table 4.2 summarizes these "enabling features" with an eye to understanding what additional mitigation would be needed to prevent successful attacks.

"Bypass ASLR, NX, Stack Canary" is an attack that is able to bypass those protection mechanisms [Shr18a]. Overwriting the GOT table means being able to change the values of the addresses within the GOT,leading to a change in execution flow based on what addresses were changed [Shr18b]. Return2libc to bypass the non-executable bit involves utilizing the included library, libc, to execute code in libc since it still allows execution [Shr18d]. Finally, using Mprotect allows the bypassing of the non-executable bit and can make the stack executable. It can do this because mprotect allows the changing of the permissions of an area in memory, effectively allowing execution permissions in memory that is already marked as non-executable [Shr18c].

Table 4.1 shows common protection mechanisms and their effectiveness against certain attacks. A check mark within Table 4.1 represents the attack is successful against that defense. For example, the "Overwrite GOT Table" attack is successful at exploiting code that has ASLR implemented and the non executable bit set [Shr18b]. Coupled with Table 4.2, the effectiveness of shuffling as a security mechanism can be determined. The enabling features in Table 4.2 detail what can lead to the attacks in Table 4.1 being effective. Of these enabling features, the ones that are directly related to object file shuffling are: One_gadget offset, gadget for Pop Rdi value, Address of _start function, Address of exit_got, and address of gadget Pop rdx, pop rsi.

Directly involved with ROP gadgets, "One_gadget offset", "Pop Rdi value", and "pop rdx, pop rsi", all use offsets to the gadget themselves, thereby learning the random start address. Once that address is known, the rest of the gadgets can be determined from their relative locations. As a result of shuffling, the addresses of the functions within the binary are different

from one variant to the next, making the reuse of the gadget offsets used in one version unable to be used in the same form in a separate version. The other two that are also prevented for similar reasoning are the address of the _start function and the address of exit_got. Both are mitigated as those locations are also affected by object file shuffling. Between all of the relevant enabling features, all of the attacks shown are mitigated, thereby designating object file shuffling as a potential mitigation technique against those attacks. Granted, testing needs to be done to determine the actual effectiveness of the method, but current analysis lends itself to being worth further research.

A new attack (JIT-ROP) exists that finds ROP gadgets dynamically, It requires the attacker to have access to a vulnerability that can leak a single byte at a specified memory address, one valid memory address, and the ability to continually exercise the vulnerability against the same process. The authors claim it will be successful against fine-grain ASLR (or shuffling) [SMD+13].

Chapter 5

Experimental Design and Methods

This chapter describes the process used to accumulate the data gathered during the performance analysis. Specifically, it goes into detail discussing the methods used to both learn how SPEC CPU® 2017 functions in a practical sense as well as the process used to develop the final script that was used to generate the final data set. Included is also a discussion of some of the prominent issues that were faced while working on getting useful data out of the benchmark.

5.1 Methods

For this research, the goal was to get a usable test suite that could be modified to where the linker could accept different lists of the object files used to create the executable. A Virtual Machine image of Ubuntu 18.04 was used and run within Oracle VirtualBox on a desktop. The desktop has an Intel(R) Core(TM) i7-8700K CPU @ 3.70GHz processor with 12 cores and 32 GB of RAM. The Ubuntu VM was running Ubuntu 18.04.4 LTS 64-bit as its OS with 6 of the 12 cores allocated to it and 16 GB of RAM.

Initially, a copy of SPEC CPU® 2017 was obtained for this research and was installed onto the VM along with applying all of the updates that the VM needed before doing any testing. At the start, testing dealt with learning how the SPEC Benchmark worked and how to use it. An initial configuration file, test.cfg, was created as a copy of the SPEC CPU® 2017 supplied Intel x86 config file. This config file is located within the /cpu2017/config folder and is where the changes that needed to be made specifically to the config file would occur. Within the config file, there are a myriad of options and flags you can edit to adjust how you want the test suite to handle each of the different executables. The primary changes concerning this research within the config file dealt with adjusting the flags to include PIC/PIE and some stack protector options. For these, the line changed was line 237: ‘‘`OPTIMIZE = -g -O3 -march=native`

`-fno-unsafe-math-optimizations -fno-tree-loop-vectorize''`. Modifications to this line allowed the incorporation of options that could be tested. For this research, these options were –no-pie, -fno-pie, –pie, -fpie, –fstack-protector and –fno-stack-protector. The reason why this specific part of the config file was edited is due to its role as the default option for all of the test suites. There are additional areas in the config file that only affect certain test suites but weren't changed in order to keep testing consistent between test suites. With the way SPEC CPU® 2017 is laid out, there are 4 different test suites to choose from to do tests with: intrate, intspeed, fprate, fpspeed. Only 2 of the 4 test suites were used for this research, intrate and intspeed. Fprate and fpspeed were unused due to initial difficulties in getting them to run successfully with base options. Fprate was used for some preliminary testing as it was able to be run without issue. Unfortunately, due to an unresolved issue, fpspeed wasn't able to run on the VM even with the default settings provided by the tutorial. (SPEC Benchmark also requires using a specific command to get access to the macros provided. This command was: source shrc.) This command gives the ability to use the other commands supplied by SPEC which includes the main command used for running each of the tests: ``runcpu --config=$BENCHMARK1config --iterations 3 --reportable $BENCHMARK1''.

In the initial stages of research, we evaluated experiments that dealt with different numbers of copies simultaneously being run. This dealt only with the intrate test suite as intspeed is dependent on the number of threads. The number of copies is set using the –copies option. From the results from SPEC CPU® 2017 on their manual page, it seemed to suggest using the number of copies as the amount of cores on the system which in this case was 6. As a result, the initial stages of experimentation included a variety of values for the –copies option, including 3, 4, 6, and 12. However, with each additional copy, the time required for each run to complete grew exponentially. For a two iteration run with 6 copies, it was already taking over 6 hours per run to complete. Add in an additional iteration and it took roughly 10 hours to complete. Granted, these results are specific to the system used so other hardware setups may produce different amounts of required runtime. Once those runs were completed, a single copy with an extra iteration per run was chosen to be the method in which all of the following tests would follow, as it allows for relatively quicker runs with adequate data to use as a baseline. From there, the next set of changes came with the modification of the Makefile.defaults file. This file is located within the /cpu2017/benchspec folder and is responsible for using the config file as a guide for determining how each of the executables are compiled and linked together.

Since the experiment was primarily concerned with shuffling the location of the object files in memory during linking time, the location within the Makefile.defaults file where the list of object files was used needed to be located. This was found on line 335: `$(EXEBASE): $(OBJS)`. For modifying things in this file, there were a few rules that were followed in order to not corrupt the makefile and ensure that it would still run properly. The first was that editing existing variables was off limits. This meant that any variables that already existed in the makefile itself couldn't be modified in any way. This meant that the `$(OBJS)` variable couldn't see any changes of any sort. The reasoning for this relates to not breaking the core functionality of the makefile. Additionally, not changing any of the variables makes it easier to replicate and reproduce the results of each test. The second rule was ensuring that only functionality native to GNU make or functionality already built into the makefile itself was permitted. This meant not using outside python scripts or something similar. Much like the reasoning behind not changing existing variables, this was done in order to preserve the integrity of the makefile in order to not cause issues in the future. The second rule ended up causing quite a bit of issues, especially when trying to create functionality that would "randomly" change the order of a list.

The strategy going into tackling this problem was to append a set of ASCII characters to each object file in the list and then sorting the list afterwards. The first attempt at this was to use the built-in function called `addprefix()` which takes text and a list and appends that text to each item in the list. Unfortunately, `addprefix()` can only handle one prefix and it gets applied to every single item. When sorted, this would lead to exactly the same list as sorting the original list normally. The next attempt revolved around using the `foreach()` function and applying one new "prefix" for each item in the list. The first few attempts at this failed due to only one of the lists included getting expanded, either the list of prefixes or the list of object files. The list that didn't get expanded was just tacked on at the end of the function. While working on the `foreach()` function, there was a realization that the prefixes would need to be removed after adding them to the list of object files. This was discovered after some confusion resulted from wondering why valid lists with the prefixes weren't working properly when the benchmark was executed. The solution to getting the prefixes added to the each item in the objects list was the `join()` function. As the name implies, the `join()` function takes two lists and joins them together, leaving any extra items in either list unjoined. Using `join()` and the functions `words()`, which returns the number of words in a list, and `wordlist()`, which returns a list based on the supplied indices, the list of the object files with the prefixes attached

was successfully created (In this case, the indices used were 1 and the number of objects in the objects list). This way, if the supplied list of prefixes was longer than the list of object files, the remaining prefixes at the end of the list would be removed by `wordlist()`, neatly combining the two lists (Currently, there is still an error if the amount of supplied prefixes being less than the number of object files). With this new list, the sort function works as intended and sorts based on directory first, thereby sorting the list by the supplied prefixes. The final function ended up as: `$(sort $(wordlist 1, $(words $(OBJS)), $(join $(PREFIXLIST), $(OBJS))))`.

The initial thought for the issue of removing the prefixes was to adapt the `notdir()` function. `Notdir()` removes up to and including everything before the last "/" and returns just the filename. This resulted in the adaptation of a "/" at the end of each prefix to feign a directory. This way, the prefixes are able to be removed. From the initial testing with a test makefile, this appeared to work as expected. However, when implementing it into Makefile.defaults, it failed to execute. From additional testing of the test makefile, it was determined that it was likely an issue caused by relative paths within the list of object files. This reasoning was sound because `notdir()` would remove the added prefix along with the path that the object file had, thus making the object reference incorrect.

This issue presented one major roadblock: there wasn't a single built-in function that could remove a specified pattern within each item of a list. There is substitution, `subst()`, and pattern substitution, `patsubst()`, but both of these functions are unable to be used for different reasons. `Subst()` cannot match with a wildcard. Therefore, the "random" prefixes wouldn't be able to be matched. It also only functions with a single supplied text and isn't able to be used with a list. `Patsubst()` can be used with a list but presents other problems. For example, it can use a single wildcard but it does a full match within a list. So if there is a pattern that needs to be matched within each individual item in the list, `patsubst()` won't work. What the solution required was the functional opposite of the `addprefix()` function: remove the specified chars from the beginning of each of the items in a list. Additionally, it needed the added functionality of working with any supplied prefix. That is, it must be able to match the chars provided for the initial sort. For this research, these characters are 'a' through 'f' inclusive, so there wouldn't be too much variance, but getting it to work for the general case would be ideal.

The solution to this ends up requiring 3 steps. The first step is to grab the prefix chars from the joined list. This is done by first taking the object with the prefix and then, using the `subst()` function, replace the "/" with a space, leaving a space delimited list of directories followed by

the object name (ex. /abc/test/testing3/object1.o becomes abc test testing3 object1.o). This is important because it allows the use of the `firstword()` function which grabs the first word in text. In the previous case it would be *abc*, which is the value that was needed. The second step is to use the first step in combination with the `foreach()` function to apply its logic to every item in the object list. The third step is to then apply the `subst()` function in combination with the `foreach()` function with the newly gained prefix to replace the prefix and final "/" with nothing for all of the object files in the list. The final version of the function is:

```
‘‘OBJS_RAND = $(foreach obj, $(sort $(wordlist 1, $(words $(OBJS)),
$(join $(LISTRAND2), $(OBJS)))), $(subst $(foreach F, $(obj),
$(firstword $(subst /, ,$F)))/,, $(obj)))’’
```

With this entire issue solved, the only issue remaining for the random sort is to supply the function with the list of prefixes that are going to be appended and the list of object files. The result is a "randomly" sorted list of the originally supplied object file list which can then be used within the makefile.

A visual representation of the effect that shuffling has on a benchmark expansion are shown in Fig. 5.1. It shows how a benchmark can be artificially modified into different versions just be incorporating object shuffling. Additionally, Fig. 5.2 displays how this affects the test suite as a whole.

In order to prove that all of these functions were working as expected when used with the benchmarks, `objdump` was used on one of the benchmarks with each of the shuffling variations. If the addresses of the functions within the output of `objdump` are located at different addresses, then the shuffling is working as expected. The benchmark used for this test was `500perlbench_r` in the intrate test suite. Fig. 5.3 shows the result of this analysis. The function "PerlIOVia_fileno" was selected at random. As shown in Fig. 5.3, the top-left is the unsorted, original benchmark, the bottom-left is the sorted variant, the top-right is the reverse sorted variant, and the bottom-right is one of the "random" variants. With each at different addresses, it shows that shuffling is working as expected.

5.2 Testing

Regarding the number of tests that were conducted, the need to establish a baseline for each test suite was imperative. After observing the length of time needed to do each test, the testing was limited to just the intrate and intspeed test suites.

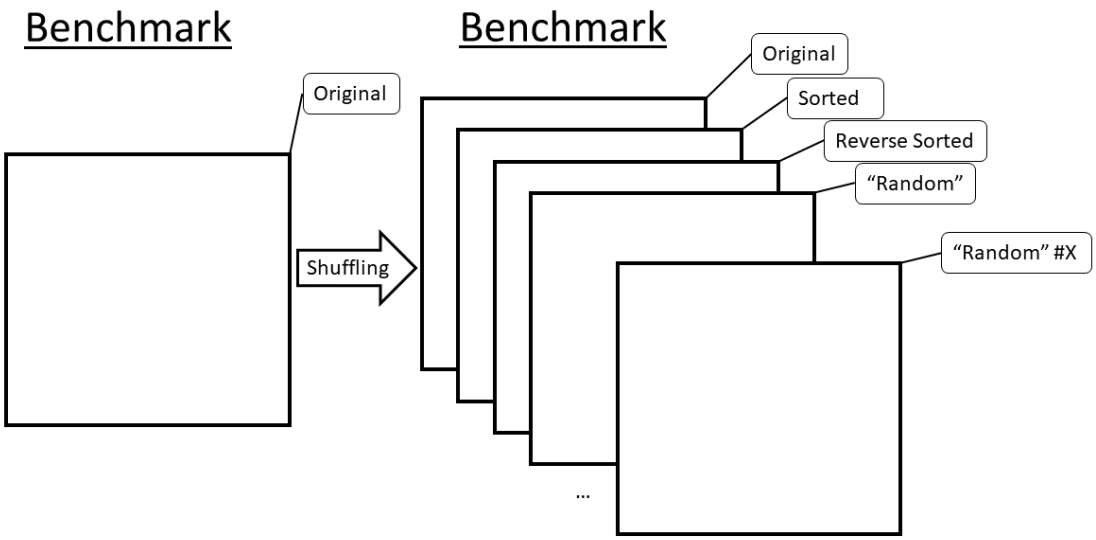

Figure 5.1: Visualization of Benchmark Expansion

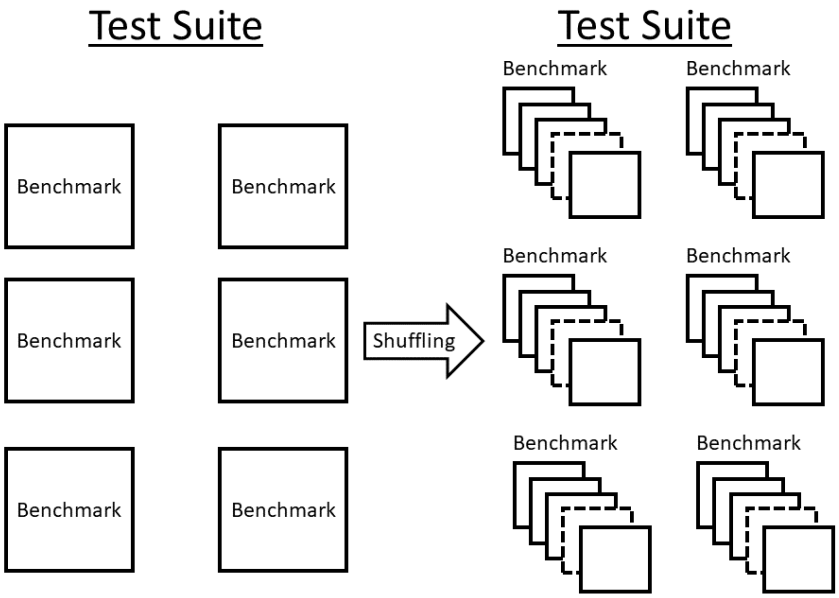

Figure 5.2: Visualization of Test Suite Expansion

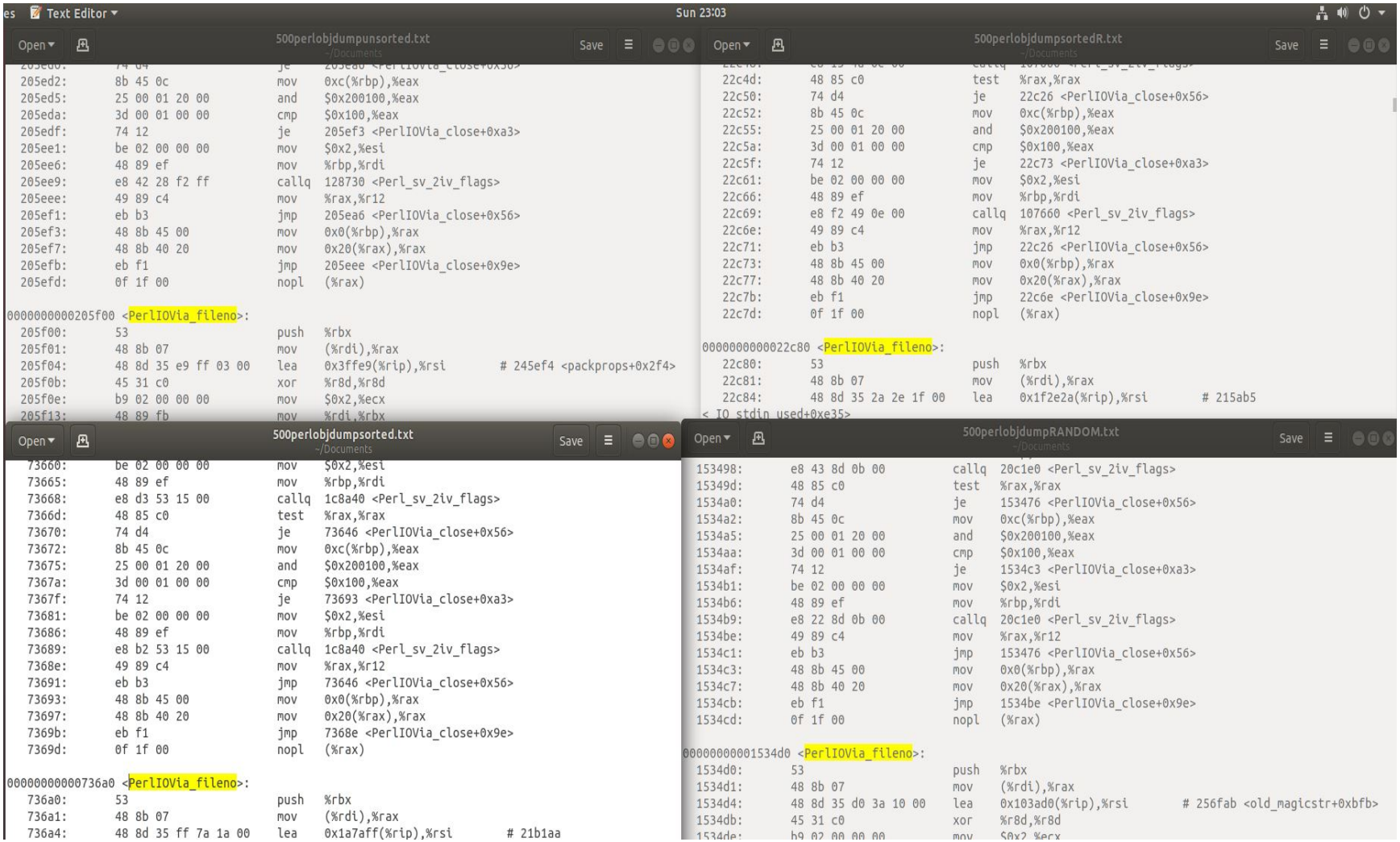

Figure 5.3: Object File Shuffling Addressing Changes

For intrate, the supplied command was:

```
‘‘runcpu --config=intrateconfig --iterations 3 --reportable intrate’’
```

For intspeed, the supplied command was:

```
‘‘runcpu --config=intspeedconfig --iterations 3 --reportable intspeed’’
```

The number of copies for intrate was set to the default 1 and the number of threads for intspeed was set to the default 4. The baseline ended up being 3 “raw” runs and 3 sorted runs in order to determine how the benchmark function normally and compare it to a sorted run, which is technically a shuffled run, to gain some insight before doing other tests. Once the baseline values were established, the main bulk of testing was initialized. This would include 1 raw run, 1 sorted run, 1 reverse sorted run, and 6 “randomly” sorted runs. In this way, there would be measurable variance between the locations of functions within each executable that would then give meaningful results once each run was completed. These runs were done for each test suite. In total, the main bulk of testing was 20 runs, 10 for intrate and 10 for intspeed. There were also additional tests on the –no-pie option as well as the –fstack-protector and –fno-stack-protector options to get some comparison on the impact they have on each of the benchmarks within the test suite. For the stack protector specifically, there was one run of raw for each of the options as well as one run of sorted which added an additional 4 runs.

In order to ensure every run was equivalent and there was no reuse of built binaries, the –action clobber option was used to destroy anything that had been created by the specified run. This way, each run would start from scratch and rebuild the binary each and every time. The –noreportable option was also included for this to leave less files at the end.

5.3 Experimental Script

Due to the amount of runs needed, their duration to execute, and the need to generalize the process, a script was developed. This script runs all of the different tests while maintaining that each of the runs had their required pieces to function correctly. This involved juggling both the unique Makefile.defaults files as well as the different config files necessary for each run. While the current version of the script requires quite a few files that aren’t included in the SPEC installation, the addition of each isn’t complex and only requires a few extra files to be added with minor changes. The script runs through each of the different runs in the following order: Raw, No PIE, Sorted, RSorted, Rand1, Rand2, Rand3, Rand4, Rand5, and Rand6. The Raw run is what all of the future runs would be compared to as it’s the

run without any changes with the original Makefile.defaults and an unedited config file. No PIE had the exact same Makefile.defaults as the Raw run but with an added option added to the config file to include the options of –fno-pie and –no-pie. The sorted run consisted of a modified Makefile.defaults to the `EXEBASE: $(OBJS)` line to `EXEBASE: $(sort $(OBJS))` and had the same config file as the Raw run. RSorted had a similar situation but used a special function to incorporate a reverse sort on the list of objects. Rand1 through Rand6 runs all used the newly developed functionality to incorporate a random sort of the list of objects with a predetermined set of prefixes to randomize by. In all, there were 10 full runs for each test suite that are executed along with their accompanied "—action clobber" runs to ensure that all of the runs remain distinct and separate. One full run of the script took approximately 84 hours or 3.5 days. There does seem to be an issue with the script itself as it has some odd output during its execution that wasn't seen during the baseline test. Fortunately, upon examination of the results, the anomalies don't seem to have an impact on the test results. The full script is provided in Appendix A.

5.4 Results Reporting and Data Accumulation

Via the included –reportable option given to each of the tests, each successful conclusion of a test outputs several different file types of the end results. There are several different values that the SPEC Benchmark assigns to completed tests including a final score called SPECrate2017_int_base for the intrate test suite and SPECspeed2017_int_base for the intspeed test suite. What was also included was the number of seconds and the ratio of each of the benchmarks. For SPECrate, seconds refers to the amount of time it took to run from the start of the first copy to the end of the last copy in seconds. For SPECspeed, seconds measure the amount of time each benchmark took to run [Cor20]. For the purposes of this research, the seconds measurement was used for comparison purposes as it is easy to understand and straightforward.

Chapter 6

Performance Analysis

This chapter analyzes the data gained from the processes and methods used in Chapter 4. It analyzes the results from both the baseline tests as well as the final script tests and determines what impact shuffling the object files has on overhead and other factors in relation to the SPEC 2017 Benchmark and its testing functionality. Also discussed is how some outside factors may have resulted in some erroneous data being generated and how they were dealt with.

6.1 Measuring Overhead and Performance Impact

This section focuses on the analysis of the results of the performance impact from each of the different tests. Shown and discussed are the initial runs, the experimental script runs, and an additional run of the script using a second system. The second system was using an Intel i7-9700 CPU @ 3.00GHz processor with 8 cores, and 16GB of RAM. This system ran the experiments on a non-VM Ubuntu 18.04 OS. Additionally, some potential explanations are provided for the difference between data sets.

6.1.1 Initial Runs

In order to get a good baseline established and to obtain a good grasp of SPEC CPU® 2017, a few test runs were completed for both the intrate and intspeed test suites. These initial tests included running the test suites with their original configuration several times. More specifically, this consisted of running the raw, original test 3 separate times along with running a sorted version of the object files. Each of the runs consisted of 3 iterations which put 9 total data points for the raw runs and 9 data points for the sorted runs.

Fig. 6.1 shows the normalized amount of seconds that the executable took to execute. The error bars are the standard deviation of each of the normalized runs. The values of the sorted runs were averaged in one of the set of graphs while median was used for the second set of

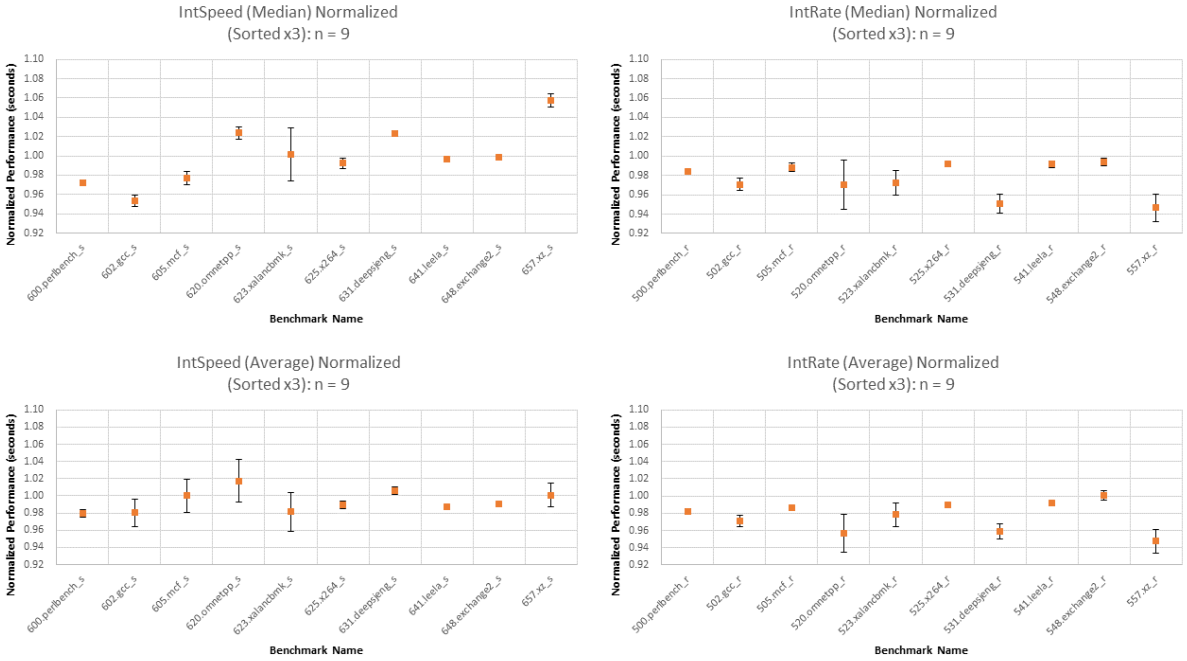

Figure 6.1: Initial Runs Graphs

graphs. Each of the runs first was calculated between its own iterations before being averaged with the rest of the runs as a means of providing a means to view the end results of each separate run. As shown in Fig. 6.1, each of the different benchmarks vary from one another by a decent percentage. In general, the amount of variance was roughly $\pm 6\%$ across all of the benchmarks for both intrate and intspeed. What this data shows is that simply changing the order of the object files can affect the performance of an executable by a measurable amount. Surprisingly, a simple shuffle can also improve the performance of executables. This is made clearly apparent by viewing the values of the intrate benchmarks in Fig. 6.1 where all of the benchmarks show at least a small improvement in performance when using the sorted variant. The intspeed benchmarks were much more varied benchmark to benchmark but with a noticeable negative impact on some of the benchmarks such as 620.ommnetpp_s. The standard deviation is also consistent at $< 1\%$, save for a few outliers like 623.xalancbmk_s. As such, this data lends itself to the conclusion that, depending on the executable, simply changing the order of the object files causes mixed results when measuring the speed of execution while the work per unit of time seems to improve.

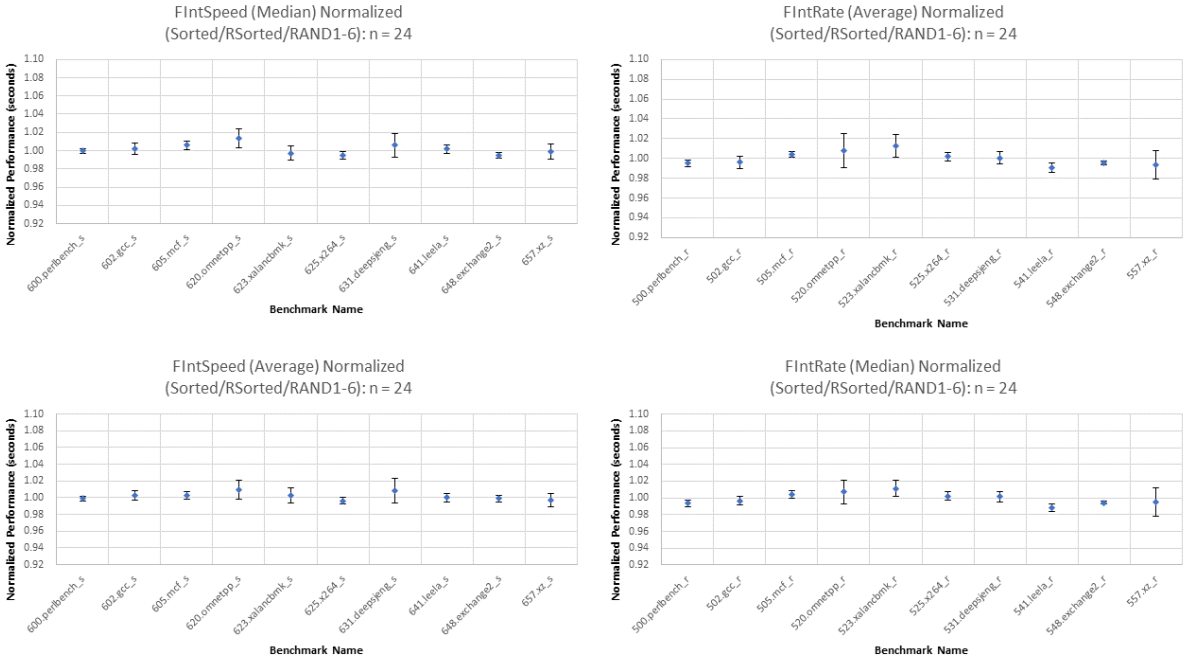

Figure 6.2: Experimental Script Results Graphs

6.1.2 Experimental Script Results

After the conclusion of the base testing with just the original test suites and their sorted version, a script was created and utilized several other variations of object file shuffling that would provide more valuable data than purely sorted object files. These variations including sorted, reverse sorted, and "random". For time efficiency, the amount of tests used per run changed to a single test run with 3 iterations per variation. There were 3 iterations of data for the raw as well as the variations including 6 different orders for the random sort variation. At 3 iterations each, this had a total of 24 total iterations of data gathered and normalized against the 3 raw runs. The reasoning for combining sorted, reverse sorted, and random variations as the same group for analysis is that they are all technically variations of a "random" sort. In other words, they can all be represented using the randomized function developed for this research.

Fig. 6.2 shows the results of the 4 day long run of the full script being completed. The one run of no-pie that was included in the script was not included in this data as it wasn't data gathered for the same purpose. The resulting data is much more consistent than the initial runs were, going from a ±6% impact to ±2% impact on the performance. Much like the initial intspeed results shown in Fig. 6.1, benchmarks have mixed results regarding the impact of object

file shuffling. In some benchmarks like 548.exchange2_r, the performance was better than that of the raw runs. On the other hand, 620.omnetpp_s shows a noticeable overhead. While not as pronounced as the initial runs, there still was an overall change. Another observation of this data is the standard deviation was across the board much smaller for these runs compared to the initial.

One of the main reasons behind the differences between the initial runs and the script may lie outside the testing environment. When doing the experiments with the script, both the desktop and the VM were devoid of any processes. Even Wi-Fi was disabled to ensure there would be no potential outside interference. This is unlike the initial runs as Chrome and other processes were running on the host machine while the VM was doing the tests. Therefore, a side effect of having programs running on the host machine may have had an impact on the results of the initial testing of the benchmarks which would explain some of the anomalies present in the initial runs that were absent in the script runs. A second potential explanation of the data is that the sorted variation in particular may have greater impact on the performance when measured alone in comparison to being included with the other variations. As such, when it is combined with the other results, it evens out to much less spiking. After analyzing the intrate runs, the first explanation is more probable due to the performance of the benchmarks going from better across the board to very mixed results.

6.1.3 Base Machine (No VM)

In order to test the validity of the runs and to test if the scripts worked in a separate environment, we ran the script on a different machine and sent the results that are shown in Fig. 6.3. Much like the original script results, the variation in performance is at around $\pm 2\%$ except for one benchmark. Interestingly, the results from this run of the script shows a much larger impact on the benchmark 623.xalancbmk_s in the intspeed test suite. Unlike the other benchmarks, it has a noticeably positive impact on the performance. Another interesting observation of the base machine results are a general decrease in the standard deviation of each benchmark meaning more consistent data was being output. Even though the impact may be small, some benchmarks from the script run and these runs had different impacts on performance. For instance, the benchmark 605.mcf_s on the first test script runs had overhead while on the base machine the runs had better performance. One explanation of these results is that regardless of the method, there always exists a sort of random performance impact depending on unknown

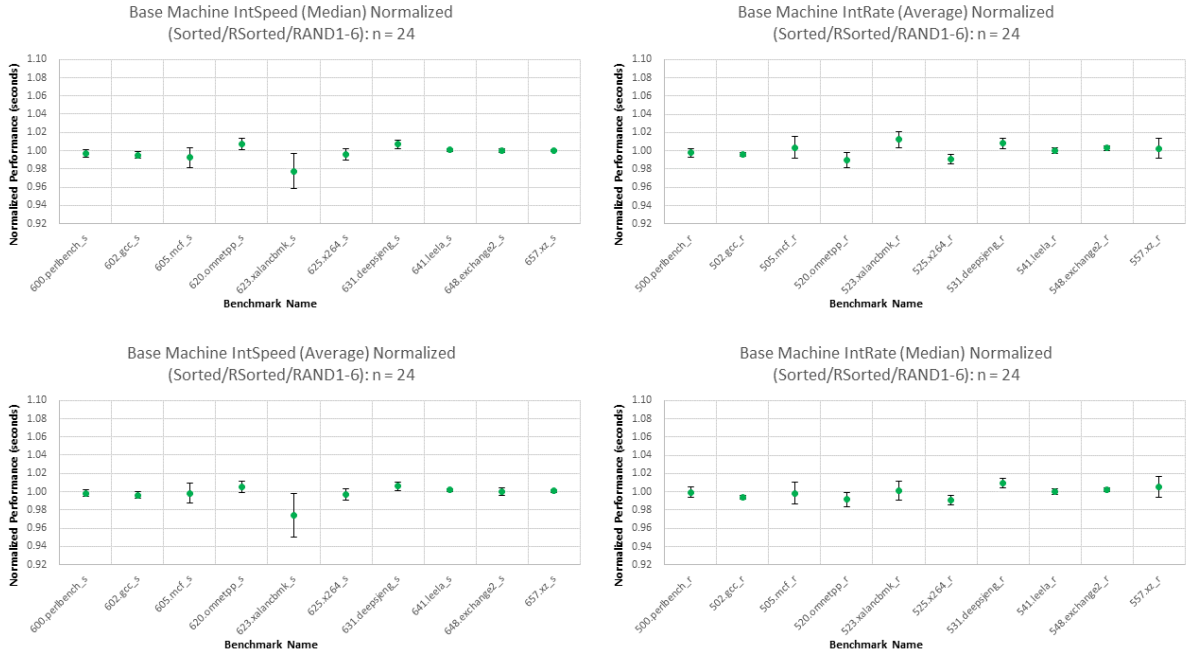

Figure 6.3: Base Machine (No VM) Script Results Graphs

outside factors during testing including exact CPU architecture, machine configuration, background processes, etc. In other words, regardless of the method that you are testing, there is a portion of the performance impact due to this "random" factor, or "noise" in the experiment. We have noticed very little discussion of this background in published security papers.

6.1.4 Threats to Validity

Due to the limited amount of tests, there are a few threats to the validity of this research. The first and foremost is the compiler options provided in the config file used for testing. The experimental script results on the VM used the x86 config file while the base machine used the x64 config file. Both of these files were the base config files provided by SPEC CPU® 2017. As such, the default optimization level provided in that config file is the basis of the results. If the optimization level is changed to a different level, changes in behavior and results done with exact same cases may end up completely different.

Another issue is the number of object files an executable uses. If an executable has roughly 10 or more object files, a reasonable number of variations can be made that are widely different from each other. However, if the amount of included object files is low, shuffling becomes much less effective as the amount of variations decreases substantially and the method ends up completely ineffective at even being considered. Granted, if its known beforehand and

the implementation of shuffling is needed or requested, creating enough object files to improve variation is a relatively simple task depending on the functionality of the executable in question.

6.1.5 Summary

Overall, we found that shuffling results in an average of ±2% performance change with a maximum increase of 2%. If shuffling is used in embedded control systems with hard real-time constraints, these variations must be taken into account prior to deploying this method.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In conclusion, there were many topics covered in this thesis, including current firmware security methods, major components involved with object file shuffling, analysis of the potential security impact of shuffling, measured performance and use of SPEC CPU® 2017, measured performance impact of object file shuffling on SPEC CPU® 2017, and development of an automated test script to run SPEC CPU® 2017.

7.1.1 Security Implications of Shuffling

Attacks were analyzed to determine their root causes in order to evaluate if shuffling could theoretically serve as a mitigation technique. These attacks ranged from bypassing ASLR and the non-executable bit using return to libc attack. Due to shuffling having an effect on the locations of functions within memory, it could be effective at stopping ROP gadget chains from being effective. ROP gadget chaining relies on assembling a list of addresses to each of the ROP gadgets that form the exploit. As such, a shuffled variant of the firmware image that has different addresses for those gadgets will cause a ROP gadget chain that was developed for a separately shuffled version of the firmware to not function correctly. This is because "gadget 1" at address 0x80932900 won't necessarily be at that address in the other version. As a result of this characteristic of shuffling, the five attacks covered are successfully stopped due to various enabling features being mitigated.

7.1.2 Technique Development for Shuffling

A simple sort and a reverse sort function were used for the initial runs of testing the capability of rearranging the order of object files. Sort and reverse sort both sort based on the file path followed by the file name. After successfully testing sort and reverse sort, a technique was

developed using built-in functions of GNU Make that takes a list of object files and a prefix list and joins them together. Since the sort function determines order based on the path first, a prefix can be created and appended to be used as the key for each object file, thus giving it a different position in the list. The prefix format used for this research was 3 alphanumeric characters between 'a' and 'f' inclusive followed by a '/'. The object files were then sorted based on the alphanumeric values in the prefix, essentially allowing any ordered list of object files. These prefixes were "randomly" created to generate a randomized list of the object files to be compiled. Once the sort was complete, the prefixes were then removed using a combination of the `subst()` and `foreach()` functions to ensure that each and every prefix was removed without harming the original object file path. To prove that this was working correctly, test cases were developed to prove that it could take potentially any set of characters as a prefix and still work properly. The reason why 3 characters were chosen is because its a small addition that is easy to "randomize" as well as being obvious on how its being sorted. Currently, the only downside to the functionality of this technique is that it needs an adequate number of prefixes in the list. The number of prefixes must match or be greater than the amount of object files. If there are fewer prefixes than object files, the technique will fail and the compilation will not take place.

7.1.3 Performance Analysis of SPEC CPU® 2017

Using the originally obtained SPEC CPU® 2017 defaults and an edited Makefile.defaults with the object files being simply sorted, performance testing was completed. There were 3 full runs of both the raw versions and the sorted versions. The sorted version results were then normalized to the raw runs to show a relative change in performance. The data that was gathered was the number of seconds each run took to complete. For each run, there were 3 iterations done for a total of 9 data points for both raw and sorted variants. From the resulting data, it showed varying increases and decreases in performance on a per benchmark basis. The general performance impact was $\pm 6\%$ for both the intspeed and intrate test suites. Surprisingly, the sorted intrate test suite had better performance across the board compared to the raw runs.

7.1.4 Performance Impact of Object File Shuffling on SPEC CPU® 2017

The experiments continued with the use of the developed script that automated the testing process for the intspeed and intrate benchmarks. A total of 10 runs were completed for each test suite: 1 raw run, 1 no-pie run, 1 sorted run, 1 reverse sorted run, and 6 variations of

"random" runs. Each run had 3 iterations to give more data for a total of 24 non raw data points. After normalizing the data similarly to the initial performance analysis, the results showed a slightly different picture than what the initial testing showed. The most obvious observation was the variance dropped to ±2% from ±6%. Also of note was that there wasn't a definitive increase or decrease across all of the benchmarks. Even though the changes were minor, there were still instances of both better and worse performances. Another run of the automated script took place on the base machine and showed similar results. The only major difference between the two data sets was the benchmark 623.xalancbmk_s in the intspeed test suite showed about a 2% increase in performance. What the data seems to conclude is that the performance impact of shuffling is minor and varies benchmark to benchmark, not having a definitive impact for better or worse in the general sense. From comparing the automated data sets to the initial runs, it seems that host processes could affect the benchmark measurements given the large decrease in variance. As such, the variance itself may be due in part to random "noise" in the system such as other active processes.

7.2 Future Work

There are many potential paths where this research can be further developed. These include creating additional randomized prefix lists for the script, additional testing of current lists to gain greater statistical significance, generalizing the script to work with command line options and environmental variables, testing common security methods such as stack protector on shuffled versions of executables for their performance impact, test the enabling features detailed in Chapter 5 for their effectiveness, and improving the script to aggregate the data and provide meaningful output that can be graphed.

7.2.1 Additional Prefix Lists

The creation of additional prefix lists will allow for greater testing and more suite expansion. It would also allow for more insight on the impact of function shuffling or other security methods. This is easily done by using the current function prefix list and generating additional lists. Currently, the function can handle any number of characters as a prefix as long as it ends with a "/". Therefore, larger or more descriptive prefixes could be used. A more interesting improvement could be made to this by generating the prefixes automatically based on the number of object files. This way, there would never be an the issue of having too few prefixes

and the script failing. It would also greatly shorten the length of a list that needed to be generated by hand.

7.2.2 Greater Statistical Significance

Due to the limited time frame, a relatively small amount of runs at 24 total data points were completed. In order to gain greater statistical significance, more runs should be done using the original tests. This way, there is more data to reinforce the observations and conclusions made during the analysis. So, an additional 10 to 15 full runs of the experimental script on the testing machine should yield enough data to gain statistical significance.

7.2.3 Script Generalization

This improvement is centered around making the script more generalized. This involves changing the current variables to be command line options or something similar. The most difficult part of generalizing the script is the generation or modification of each of the different Makefile.defaults needed for each of the separate runs. The addition of the specified runs you wanted to use would also be a very important addition. This would look something like ‘‘`runscript 3 -bchmks intspeed,intrate -sorted 3 -random1 6`’’. This command would then run 3 raw runs, 3 sorted runs, and 6 random1 runs for both intspeed and intrate test suites. Additional testing to get fprate and fpspeed to work would also be helpful for this direction.

7.2.4 Security Method Testing

This direction would consist of testing the implementation of common security methods on shuffled versions of executables. This was already done to a minor degree with non executable stack but wasn't included as it wasn't extensive enough to warrant its inclusion. Some security methods that would be good tests would be ASLR, non executable stack, and pie/no pie. These tests would be possible through the modification of the config file used to add those specific options.

7.2.5 Enabing Features Testing

Testing the attacks listed in Table 4.1 would go a long way in determining the practical impact of object file shuffling on security. As it has only scratched the surface of research, this would be imperative to learning how useful it can be. A way to go about testing this would be to first test each of the benchmarks and determine if they are vulnerable to any of the attacks listed. If they are, then create a shuffled version of that benchmark and attempt the same attack. If

the attack is successful, object file shuffling was ineffective and vice versa. Many in-depth test cases would need to be developed to ensure that object file shuffling is fully examined for its practical security applications before being deployed.

7.2.6 Script Data Aggregation

As it currently stands, the data output by each run of the script has to be manually collected and graphed. However, one of the output files for the results is a .csv file which can be used to grab the data after every run. Once the data is grabbed, it can then be used after all of the runs have completed to combine into one area that can then be graphed. The main difficulty in getting this functionality to work properly would be finding the correct results file and gathering the correct data. Once that has been accomplished, its a simple matter to apply normalization and get the standard deviation. All that would be needed would be to graph the results.

7.3 Related Work

There are a couple of other related works dealing with related topics. First, there is the research on the performance impact of position independent executables (PIE) on SPEC CPU® 2006 [Pay12]. Secondly, there is dynamic shuffling via a tool called *Shuffler* [WKGWK+16].

7.3.1 Position Independent Executable (PIE) Impact

To determine the performance impact of using position independent executables, a study was done using SPEC CPU® 2006. The system they used for testing was an Intel Core i7 dual core CPU at 3.07 GHz and 12 GB of RAM. The OS that was used was Ubuntu 11.04 with GCC 4.5.2-8ubuntu. They conclude that PIE has up to a 26% overhead on individual benchmarks with an average overhead of 10% with their reasoning being increased register pressure as the addition of PIE reduces available registers from 6-7 to 5-6 [Pay12]. Since their results were using SPEC CPU® 2006, it is difficult to draw direct comparisons. However, it would be worth the additional testing of PIE and No-PIE options to determine if similar results can be obtained.

7.3.2 Dynamic Shuffling via *Shuffler*

Shuffler by Williams-King et al. [WKGWK+16] is a different approach to shuffling where, instead of using a static shuffling method, it utilizes dynamic shuffling by randomizing the code locations while the program is running. The way they accomplish this ability is by having *Shuffler* working on the modification of the code asynchronously via a separate thread. Once it has

been re-randomized, it then swaps to the new copy. Their results include several interesting observations. First, the tool is effective at preventing attacks such as return-oriented-programming (ROP) and just-in-time return oriented programming JIT-ROP(). The system they ran their tests on had a Westmere Xeon X5660 processor at 2.8GHz with 24 cores and 64 GBs of RAM. The OS they used was Ubuntu 16.04 with GGC 4.8.4. Additionally, they reported an overhead of 14.9% using SPEC CPU® 2006. They also require at least twice the amount of RAM (to create the copy) and a free CPU thread to perform the copying. For our purposes, the amount of overhead that Shuffler imposes on the benchmarks is a heavy negative as the focus of the research was prioritizing the limiting of any overhead, especially for systems with limited resources.

Bibliography

[AAB+17] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. Understanding the Mirai Botnet. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1093–1110, Aug 2017.

[Ale96] One Aleph. Smashing the stack for fun and profit. *http://www. shmoo. com/phrack/Phrack49/p49-14*, 1996.

[BBLD13] Zachry Basnight, Jonathan Butts, Juan Lopez, and Thomas Dube. Firmware modification attacks on programmable logic controllers. *International Journal of Critical Infrastructure Protection*, 6(2):76 – 84, 2013.

[CBD+99] Crispin Cowan, Steve Beattie, Ryan Finnin Day, Calton Pu, Perry Wagle, and Erik Walthinsen. Protecting systems from stack smashing attacks with StackGuard. In *Linux Expo*, 1999.

[Cen20] IBM Knowledge Center. When to use dynamic linking and static linking, 2020.

[CFBX11] Ping Chen, Yi Fang, Mao Bing, and Li Xie. JITDefender: A Defense against JIT Spraying Attacks. In Jan Camenisch, Simone Fischer-Hübner, Yuko Murayama, Armand Portman, and Carlos Rieder, editors, *Future Challenges in Security and Privacy for Academia and Industry*, pages 142–153. Springer Berlin Heidelberg, 2011.

[Cor20] Standard Performance Evaluation Corporation. SPEC CPU® 2017, 2020.

[CT91] Steve Chamberlain and Ian Lance Taylor. The GNU linker, 1991.

[EPAG16] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

[FCS+15] Alessandro Di Federico, Amat Cama, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. How the ELF ruined christmas. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 643–658. USENIX Association, Aug 2015.

[Fre09] Free Software Foundation, Inc. *gcc Linux User's Manual*, 2009.

[LHG+11] Limin Liu, Jin Han, Debin Gao, Jiwu Jing, and Daren Zha. Launching Return-Oriented Programming Attacks against Randomized Relocatable Executables. In *2011 IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 37–44, 2011.

[MGR14] Hector Marco-Gisbert and Ismael Ripoll. On the effectiveness of full-ASLR on 64-bit linux. In *Proceedings of the In-Depth Security Conference*, 2014.

[MGR16] Hector Marco-Gisbert and Ismael Ripoll. Exploiting linux and PaX ASLR's weaknesses on 32-and 64-bit systems. *BlackHat Asia*, 2016.

[MSL12] Jonathan AP Marpaung, Mangal Sain, and Hoon-Jae Lee. Survey on malware evasion techniques: State of the art and challenges. In *2012 14th International Conference on Advanced Communication Technology (ICACT)*, pages 744–749. IEEE, 2012.

[Pay12] Mathias Payer. Too much PIE is bad for performance. *Technical report*, 766, 2012.

[RBSS12] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1), Mar 2012.

[Ric02] Gerardo Richarte. Four different tricks to bypass stackshield and stackguard protection. *World Wide Web*, 1, 2002.

[Sat18] Michael Satran. Data Execution Prevention - Win32 apps. *Win32 apps — Microsoft Docs*, 2018.

[SD08] Alexander Sotirov and Mark Dowd. Bypassing browser memory protections in Windows Vista. *Blackhat USA*, 2008.

[SDA02] Benjamin Schwarz, Saumya Debray, and Gregory Andrews. Disassembly of executable code revisited. In *Ninth Working Conference on Reverse Engineering, 2002*, pages 45–54, Nov 2002.

[Ser12] Fermin J Serna. The info leak era on software exploitation. *Black Hat USA*, 2012.

[SGGK07] Nenad Stojanovski, Marjan Gusev, Danilo Gligoroski, and Svein J. Knapskog. Bypassing Data Execution Prevention on Microsoft Windows XP SP2. In *The Second International Conference on Availability, Reliability and Security (ARES'07)*, pages 1222–1226, 2007.

[Sha07] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-Libc without Function Calls (on the X86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, page 552–561, 2007.

[Shr18a] Shivam Shrirao. Format String Exploits: Defeating Stack Canary, NX and ASLR Remotely on 64 bit. https://www.ret2rop.com/2018/08/format-string-defeating-stack-canary-nx-aslr-remote.html, Aug 2018.

[Shr18b] Shivam Shrirao. Format Strings: GOT Table Overwrite to Change Control Flow Remotely on ASLR. https://www.ret2rop.com/2018/10/format-strings-got-overwrite-remote.html, Oct 2018.

[Shr18c] Shivam Shrirao. Make Stack Executable Again. https://www.ret2rop.com/2018/08/make-stack-executable-again.html, Aug 2018.

[Shr18d] Shivam Shrirao. Return to libc on Modern 32 bit and 64 bit Linux. https://www.ret2rop.com/2018/08/return-to-libc.html, Aug 2018.

[SMD+13] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *2013 IEEE Symposium on Security and Privacy*, pages 574–588, May 2013.

[SPP+04] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the Effectiveness of Address-Space Randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, page 298–307. Association for Computing Machinery, 2004.

[Ven20] Varsha Venugopal. Review of modern attacks against common security mechanisms. Technical report, University of Idaho Center for Secure and Dependable Systems, 2020.

[WC+03] Perry Wagle, Crispin Cowan, et al. Stackguard: Simple stack smash protection for gcc. In *Proceedings of the GCC Developers Summit*, pages 243–255, 2003.

[WKGWK+16] David Williams-King, Graham Gobieski, Kent Williams-King, James P. Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P. Kemerlis, Junfeng Yang, and William Aiello. Shuffler: Fast and deployable continuous code re-randomization. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 367–382. USENIX Association, Nov 2016.

[ZDWZ03] Chao Zhang, Lei Duan, Tao Wei, and Wei Zou. Secgot: Secure global offset tables in ELF executables. In *Proceedings of the 2nd International Conference on Computer Science and Electronics Engineering*. Atlantis Press, 2013/03.

Appendix A

Sample Code

This chapter contains the edited code within each of the different Makefile.defaults variants. This includes the original, sorted, reverse sorted, and "random" functions. A copy of one of the test suite runs of the script is also included following the Makefile.defaults.

A.1 Original Makefile.defaults

```
##############################################################################
# TARGETS -- Rules to build benchmark executables

all: build

$(EXEBASE): $(OBJS)
```

A.2 Sorted Makefile.defaults

```
##############################################################################
# TARGETS -- Rules to build benchmark executables

all: build

$(EXEBASE): $(sort $(OBJS))
#$(EXEBASE): $(OBJS)
```

A.3 Reverse Sorted Makefile.defaults

```
reverse = $(if $(wordlist 2,2,$(1)), $(call reverse, $(wordlist 2,
$(words $(1)), $(1))) $(firstword $(1)), $(1))

################################################################################
# TARGETS -- Rules to build benchmark executables

all: build

#$(EXEBASE): $(sort $(OBJS))
$(EXEBASE): $(call reverse, $(OBJS))
```

A.4 Randomly Sorted Makefile.defaults

```
OBJS_RAND = $(foreach obj, $(sort $(wordlist 1, $(words $(OBJS)),
$(join $(LISTRAND7), $(OBJS)))), $(subst $(foreach F, $(obj),
$(firstword $(subst /, ,$F)))/,, $(obj)))


################################################################################
# TARGETS -- Rules to build benchmark executables

all: build

$(EXEBASE): $(OBJS_RAND)
```

A.5 Experimental Script

```
#!/bin/bash
BENCHMARK1="intspeed"
BENCHMARK2="intrate"
CURRENTBENCHMARK="NULL"

parent_path=$(cd "$(dirname "${BASH_SOURCE[0]}")" ; pwd -P)
echo "$parent_path"


# This section removes the current Makefile.defaults file and uses the modified
# makefiles' orignal versions to be used for each test

echo $(cp $parent_path/benchspec/Makefile.defaults
    $parent_path/benchspec/MakefileOrig.defaults)
echo $(rm $parent_path/benchspec/Makefile.defaults)
echo $(cp $parent_path/benchspec/MakefileRAWOrig.defaults
    $parent_path/benchspec/MakefileRAW.defaults)
echo $(cp $parent_path/benchspec/MakefileSORTEDOrig.defaults
    $parent_path/benchspec/MakefileSORTED.defaults)
echo $(cp $parent_path/benchspec/MakefileRSORTEDOrig.defaults
    $parent_path/benchspec/MakefileRSORTED.defaults)
echo $(cp $parent_path/benchspec/MakefileRANDOrig2.defaults
    $parent_path/benchspec/MakefileRAND2.defaults)
echo $(cp $parent_path/benchspec/MakefileRANDOrig3.defaults
    $parent_path/benchspec/MakefileRAND3.defaults)
echo $(cp $parent_path/benchspec/MakefileRANDOrig4.defaults
    $parent_path/benchspec/MakefileRAND4.defaults)
echo $(cp $parent_path/benchspec/MakefileRANDOrig5.defaults
    $parent_path/benchspec/MakefileRAND5.defaults)
```

```
echo $(cp $parent_path/benchspec/MakefileRANDOrig6.defaults
    $parent_path/benchspec/MakefileRAND6.defaults)
echo $(cp $parent_path/benchspec/MakefileRANDOrig7.defaults
    $parent_path/benchspec/MakefileRAND7.defaults)


# First tested is the raw version, using the specified benchmark config file

echo $(mv $parent_path/benchspec/MakefileRAW.defaults
    $parent_path/benchspec/Makefile.defaults)

echo "---------------- Starting $BENCHMARK1 RAW ----------------"
runcpu --config=${BENCHMARK1}config --iterations 3 --reportable $BENCHMARK1
runcpu --action clobber --config=${BENCHMARK1}config --noreportable $BENCHMARK1
echo "---------------- Finished $BENCHMARK1 RAW ----------------"

# Second tested is the No-Pie version, using the specified benchmark config file.
# It is the only test with a different config file from the same group of benchmarks.

echo "---------------- Starting $BENCHMARK1 NO PIE/PIC ----------------"
runcpu --config=${BENCHMARK1}NPconfig --iterations 3 --reportable $BENCHMARK1
runcpu --action clobber --config=${BENCHMARK1}NPconfig --noreportable $BENCHMARK1
echo "---------------- Finished $BENCHMARK1 NO PIE/PIC ----------------"

# This part is where the changing of the Makefiles occur.
# It renames the Makefile back to its previous name before the test.

echo $(mv $parent_path/benchspec/Makefile.defaults
    $parent_path/benchspec/MakefileRAW.defaults)
echo $(mv $parent_path/benchspec/MakefileSORTED.defaults
    $parent_path/benchspec/Makefile.defaults)
```

```
# Third tested is the sorted list of object files.

echo "---------------- Starting $BENCHMARK1 SORTED ----------------"
runcpu --config=${BENCHMARK1}config --iterations 3 --reportable $BENCHMARK1
runcpu --action clobber --config=${BENCHMARK1}config --noreportable $BENCHMARK1
echo "---------------- Finished $BENCHMARK1 SORTED ----------------"

echo $(mv $parent_path/benchspec/Makefile.defaults
    $parent_path/benchspec/MakefileSORTED.defaults)
echo $(mv $parent_path/benchspec/MakefileRSORTED.defaults
    $parent_path/benchspec/Makefile.defaults)

# Forth tested is the reversed sorted list of object files.

echo "---------------- Starting $BENCHMARK1 RSORTED1 ----------------"
runcpu --config=${BENCHMARK1}config --iterations 3 --reportable $BENCHMARK1
runcpu --action clobber --config=${BENCHMARK1}config --noreportable $BENCHMARK1
echo "---------------- Finished $BENCHMARK1 RSORTED1 ----------------"

echo $(mv $parent_path/benchspec/Makefile.defaults
    $parent_path/benchspec/MakefileRSORTED.defaults)
echo $(mv $parent_path/benchspec/MakefileRAND2.defaults
    $parent_path/benchspec/Makefile.defaults)

# Fifth through final tests are the random sorts.
# Each random sort has their own set of unique prefixes for object
# files that are used to then be sorted by.
# If you wish to add additional randomized prefix lists, the minimum amount
# is the amount of object files that the benchmark uses. From what I could
# tell from testing, this as around 110 prefixes.

echo "---------------- Starting $BENCHMARK1 RANDOM1 ----------------"
```

```
runcpu --config=${BENCHMARK1}config --iterations 3 --reportable $BENCHMARK1
runcpu --action clobber --config=${BENCHMARK1}config --noreportable $BENCHMARK1
echo "---------------- Finished $BENCHMARK1 RANDOM1 ----------------"

echo $(mv $parent_path/benchspec/Makefile.defaults
    $parent_path/benchspec/MakefileRAND2.defaults)
echo $(mv $parent_path/benchspec/MakefileRAND3.defaults
    $parent_path/benchspec/Makefile.defaults)

echo "---------------- Starting $BENCHMARK1 RANDOM2 ----------------"
runcpu --config=${BENCHMARK1}config --iterations 3 --reportable $BENCHMARK1
runcpu --action clobber --config=${BENCHMARK1}config --noreportable $BENCHMARK1
echo "---------------- Finished $BENCHMARK1 RANDOM2 ----------------"

echo $(mv $parent_path/benchspec/Makefile.defaults
    $parent_path/benchspec/MakefileRAND3.defaults)
echo $(mv $parent_path/benchspec/MakefileRAND4.defaults
    $parent_path/benchspec/Makefile.defaults)

echo "---------------- Starting $BENCHMARK1 RANDOM3 ----------------"
runcpu --config=${BENCHMARK1}config --iterations 3 --reportable $BENCHMARK1
runcpu --action clobber --config=${BENCHMARK1}config --noreportable $BENCHMARK1
echo "---------------- Finished $BENCHMARK1 RANDOM3 ----------------"

echo $(mv $parent_path/benchspec/Makefile.defaults
    $parent_path/benchspec/MakefileRAND4.defaults)
echo $(mv $parent_path/benchspec/MakefileRAND5.defaults
    $parent_path/benchspec/Makefile.defaults)

echo "---------------- Starting $BENCHMARK1 RANDOM4 ----------------"
runcpu --config=${BENCHMARK1}config --iterations 3 --reportable $BENCHMARK1
runcpu --action clobber --config=${BENCHMARK1}config --noreportable $BENCHMARK1
```

```
echo "---------------- Finished $BENCHMARK1 RANDOM4 ----------------"

echo $(mv $parent_path/benchspec/Makefile.defaults
    $parent_path/benchspec/MakefileRAND5.defaults)
echo $(mv $parent_path/benchspec/MakefileRAND6.defaults
    $parent_path/benchspec/Makefile.defaults)

echo "---------------- Starting $BENCHMARK1 RANDOM5 ----------------"
runcpu --config=${BENCHMARK1}config --iterations 3 --reportable $BENCHMARK1
runcpu --action clobber --config=${BENCHMARK1}config --noreportable $BENCHMARK1
echo "---------------- Finished $BENCHMARK1 RANDOM5 ----------------"

echo $(mv $parent_path/benchspec/Makefile.defaults
    $parent_path/benchspec/MakefileRAND6.defaults)
echo $(mv $parent_path/benchspec/MakefileRAND7.defaults
    $parent_path/benchspec/Makefile.defaults)

echo "---------------- Starting $BENCHMARK1 RANDOM6 ----------------"
runcpu --config=${BENCHMARK1}config --iterations 3 --reportable $BENCHMARK1
runcpu --action clobber --config=${BENCHMARK1}config --noreportable $BENCHMARK1
echo "---------------- Finished $BENCHMARK1 RANDOM6 ----------------"

echo "------------------------------------------------------------"
echo "------------------- $BENCHMARK1 COMPLETED --------------------"
echo "------------------------------------------------------------"


# This section removes all added files that were created from their originals


echo $(rm $parent_path/benchspec/MakefileRAW.defaults)
echo $(rm $parent_path/benchspec/MakefileSORTED.defaults)
```

```
echo $(rm $parent_path/benchspec/MakefileRSORTED.defaults)
echo $(rm $parent_path/benchspec/MakefileRAND2.defaults)
echo $(rm $parent_path/benchspec/MakefileRAND3.defaults)
echo $(rm $parent_path/benchspec/MakefileRAND4.defaults)
echo $(rm $parent_path/benchspec/MakefileRAND5.defaults)
echo $(rm $parent_path/benchspec/MakefileRAND6.defaults)
```